Università degli Studi di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Redefining and Evaluating Coverage Criteria Based on the Testing Scope

Breno Miranda

Supervisor
Antonia Bertolino (ISTI-CNR)

Academic Year 2016-2017

Author's e-mail: breno.miranda@di.unipi.it

Author's address:

Dipartimento di Informatica
Università degli Studi di Pisa
Largo Bruno Pontecorvo, 3
56127 Pisa
Italia

To my parents

Abstract

Test coverage information can help testers in deciding when to stop testing and in augmenting their test suites when the measured coverage is not deemed sufficient. Since the notion of a test criterion was introduced in the 70's, research on coverage testing has been very active with much effort dedicated to the definition of new, more cost-effective, coverage criteria or to the adaptation of existing ones to a different domain. All these studies share the premise that after defining the entity to be covered (e.g., branches), one cannot consider a program to be adequately tested if some of its entities have never been exercised by any input data. However, it is not the case that all entities are of interest in every context. This is particularly true for several paradigms that emerged in the last decade (e.g., component-based development, service-oriented architecture). In such cases, traditional coverage metrics might not always provide meaningful information. In this thesis we address such situation and we redefine coverage criteria so to focus on the program parts that are relevant to the testing scope. We instantiate this general notion of scope-based coverage by introducing three coverage criteria and we demonstrate how they could be applied to different testing contexts. When applied to the context of software reuse, our approach proved to be useful for supporting test case prioritization, selection and minimization. Our studies showed that for prioritization we can improve the average rate of faults detected. For test case selection and minimization, we can considerably reduce the test suite size with small to no extra impact on fault detection effectiveness. When the source code is not available, such as in the service-oriented architecture paradigm, we propose an approach that customizes coverage, measured on invocations at service interface, based on data from similar users. We applied this approach to a real world application and, in our study, we were able to predict the entities that would be of interest for a given user with high precision. Finally, we introduce the first of its kind coverage criterion for operational profile based testing that exploits program spectra obtained from usage traces. Our study showed that it is better correlated than traditional coverage with the probability that the next test input will fail, which implies that our approach can provide a better stopping rule. Promising results were also observed for test case selection. Our redefinition of coverage criteria approaches the topic of coverage testing from a completely different angle. Such a novel perspective paves the way for new avenues of research towards improving the cost-effectiveness of testing, yet all to be explored.

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

Coverage testing has been a well-established concept since the early days of the software testing discipline. Already in the 1970's, Goodenough and Gerhart [48, 49] introduced the notion of a test criterion, i.e., the criterion that defines what constitutes an adequate test. Their initial definition required that a test criterion should be *reliable* and *valid*. To be reliable, a test criterion should guarantee the selection of tests that were consistent in their ability to reveal faults. To be considered valid, on the other hand, a test criterion should ensure that for every fault in a program, there should exist a complete set of test data capable of revealing that fault. Such an *ideal criterion* was quickly shown to be not practically applicable [62, 133], and since then, the research focus shifted from the theoretical sphere to pursuit practically applicable criteria. Only in 1988, however, there was a shift from the theoretical to the practical way of approaching the topic of adequacy criteria with the seminal work of Frankl and Weyuker [37] in which they proposed the applicable family of data flow testing criteria. Adequacy criteria proposed prior to that work could not achieve 100% coverage (because of infeasible paths, for example). After Frankl and Weyuker redefined existing adequacy criteria to cover the entities that could be reached, achieving full coverage (100%) was now possible.

In the years that followed, countless new adequacy criteria have been proposed in the literature and, even though they may target widely different entities, they are all based on the same underlying principle: *a set of entities that must be covered is identified (they could be statements, branches, paths, functions, and so on), and a program is not considered to be adequately tested until all entities (or a given percentage thereof) have been executed at least once.* Coverage is then measured as the percentage of covered entities with respect to the total number of entities in the program under test. However, for complex systems that are (re)used in different contexts traditional coverage might not always provide meaningful information. In fact, *not all entities might be of interest in every context.* This is particularly true for several new programming paradigms that emerged in the last decade: component-based development (CBD), service-oriented architecture (SOA), and cloud computing are just a few examples. Note that we do not refer here to the well-known problem of infeasible paths. We speak of entities that are perfectly feasible, i.e., there would

exist test inputs that exercise them, but such test inputs are not relevant in a usage context because this user would never invoke such inputs in operation.

This thesis addresses such situation and aims at adapting test coverage measures to the specific testing *scope*. More precisely, our proposal is to take into account the way the system under test is going to be (re)used to identify the relevance of each entity to the new scope. While of course where feasible the maximum number of entities should be covered for enhanced confidence, we claim that *if testing time and resources are limited*, then it may make sense to target first those entities that are relevant to the specific testing scope, because doing so we improve test effectiveness on the most important faults for that scope.

1.1 Context and Objectives

Bartolini and coauthors introduced in [13] the notion of *relative coverage*, by which coverage measures should take into consideration how a given service would be used by the client, i.e., *coverage measures should take into consideration the testing scope*. The work was developed in the context of service-oriented architecture and the authors claimed that traditional testing approaches should be revised to deal with service-oriented systems. This thesis is inspired by this notion of relative coverage.

Differently from the traditional way of measuring coverage (see Equation 1.1) in which the number of entities expected to be covered always embraces the full set of available entities of a given kind, the notion of relative coverage allows us to have a flexible and context-dependent number of *in-scope* entities.

$$\text{Traditional coverage} = \frac{\textit{number of covered entities}}{\textit{number of available entities}} \cdot 100(\%) \tag{1.1}$$

The equation used to calculate relative coverage is analogous to that of traditional coverage. Indeed, the only difference is that one should first identify the set of *in-scope* entities for a given testing scope before computing coverage metrics. We use the term *in-scope entities* to refer to the entities from the system under test that are relevant to a given testing scope. The remaining ones are referred to as *out-of-scope entities*.

$$\text{Relative coverage} = \frac{\textit{number of covered entities}}{\textit{number of in-scope entities}} \cdot 100(\%) \tag{1.2}$$

Computing coverage metrics taking into account only the entities that are relevant to a given testing scope is an attractive idea. However, it brings the difficult challenge of having to identify *what are the in-scope entities for a given testing scope* from time to time (i.e., what is the denominator in the coverage ratio?). In this thesis we propose and evaluate different approaches for automating the process of identifying the set of in-scope entities for a given testing scope. Additionally,

we also investigate the usefulness of relative coverage when used as adequacy and selection criteria.

At this point, two things need to be made clear. First, the motivation behind the adoption of relative coverage is *not* in merely achieving a higher score. Rather, it aims at providing a more *realistic estimate* of what could be achieved by augmenting the test suite. On this regard, we stress that coverage metrics are useful to guide developers and testers to find areas in the program that have not been exercised and they can even be used as a stopping criteria for the testing activities if a given target is defined in advance. However, they are *not* a measure of quality or correctness of the program being tested. Achieving 100% of statement coverage is not necessarily correlated to the quality of the testing performed as some paths may be missed, and even reaching 100% path coverage, which is impractical in most of the cases (because of infeasible paths or because the program may contains cycles, which would cause the number of paths to be infinite or too large, for example), may not necessarily be a useful information. If test cases are created just to increase the coverage rate without the objective of exercising error-prone areas and possibly revealing faults, achieving high levels of coverage does not help in gaining confidence about the level of quality of the software being tested.

Second, by proposing to focus the testing efforts on the in-scope entities we do *not* claim that less testing should be carried out. It is a well-known fact that exhaustive testing is impractical in the vast majority of the cases. As a result, typically some testing strategy is defined to make sure that the available resources are used efficiently. However, it is also well known that the testing activities are often penalized in the case of project time shortening. In a survey conducted by Torkar and Mankefors [124], 60% of the developers claimed that verification and validation was the first thing that was neglected in case of time shortage during a project. While of course where feasible even out-of-scope entities should be covered for enhanced confidence, we claim that *if testing time and resources are limited*, then it may make sense to target first those entities that are in-scope, because doing so we improve test effectiveness on relevant faults.

In Chapters 4, 5, and 6 we will introduce new adequacy criteria inspired by the idea of relative coverage. For each new criterion introduced, we propose a different definition for what in-scope means, which may depend on the context, on the user, or on another factor. We baptized each one of the new adequacy criteria introduced with mnemonics for ease of association with the explored testing scenario, but all of them are simply different instantiations of the relative coverage equation. More precisely, in Chapter 4 we introduce a coverage criterion tailored for the context of software reuse and we call it "*relevant coverage*". In Chapter 5, we introduce the "*social coverage*", a coverage criterion that customizes coverage information to a a given user by leveraging coverage data from similar users. Finally, in Chapter 6, we introduce a coverage criterion for test adequacy and selection for operational profile based testing and we call it "*operational coverage*".

1.2 Contributions

The main contribution of this thesis is to approach coverage testing from a completely different perspective. Here we redefine coverage criteria inspired by the idea of relative coverage. By doing so, we move from the rigid nature of traditional coverage to a flexible and context-dependent notion that customizes coverage information according to the testing scope. For each new criterion introduced here, we propose ways of automating the identification of the in-scope entities as well as the computation of the proposed relative coverage criterion. Additional contributions are provided in the next chapters as follows:

- **Chapter 4**
  - The redefinition of coverage criteria for the context of software reuse;
  - The first work specifically addressing test case prioritization, selection and minimization for reused software.
- **Chapter 5**
  - The definition of a novel notion of relative coverage that customizes coverage information to a given user by leveraging coverage information from similar users;
  - An exploratory study on a real-world system assessing the accuracy of the proposed approach in predicting the relevant entities to a given tester.
- **Chapter 6**
  - The definition of a coverage criterion for operational profile based testing;
  - The proposal of a novel method of measuring code coverage that exploits program spectra;
  - The design of the first of its kind study of using (operational profile based) coverage for test adequacy and selection in the context of operational profile based testing.

1.3 General Definitions

In this section we introduce, for the sake of uniformity, some basic concepts that will be extensively referenced throughout this thesis. For some concepts that have been already anticipated (e.g., *in-scope entities*), here we provide a more formal definition.

Definition 1: *Entity.* The term "entity" (or its plural form, "entities") is used according to the definition from the Oxford Dictionary of English — *a thing with*

distinct and independent existence — to refer to the units that are commonly used in the coverage metrics (e.g., statements, branches, paths, variable definitions, variable uses, etc).

Definition 2: (*Testing*) *Scope.* A subset of the (testing) input domain. More formally, given the input domain $\mathcal{D}$ of a program P, and given a set C of constraints over $\mathcal{D}$, a (testing) scope $\mathcal{S}$ is defined by the set of (test) input values to program P that satisfy the constraints C.

In the above definition, the constraints can be as formal as algebraic expressions over P input variables, or general properties delimiting the input domain $\mathcal{D}$. Notice that in this thesis we are not providing a general definition of the testing scope. Rather, we assume that the information regarding the specific testing context is available. We will provide examples in the developed case studies.

Definition 3: *In-scope entities.* The set of entities relevant to a given scope. More formally, given a program P with entities $\{e_1, e_2, ..., e_n\}$ and a scope $\mathcal{S}$, the set of in-scope entities with regards to S is $\mathcal{E}_s = \{e_{i_1}, e_{i_2}, ..., e_{i_n}\}$ such that $\forall e_{i_j}$ there exists some input $v \in S$ that covers them.

Definition 4: *Out-of-scope entities.* The set of entities that are not relevant to a given scope (they are not covered by any input $v \in S$).

For the purpose of exposition in the following we will refer to relevant, or in-scope, faults. This notion would refer to those faults that can be triggered by some inputs within the scope. In real life this notion cannot be easily defined since doing this would require to exhaustively exercise the program onto the whole scope $\mathcal{S}$, which is not feasible except for trivial programs. For nondeterministic programs even this would not be sufficient. Therefore the notion of relevant faults can only be defined in probabilistic terms.

Definition 5: *In-scope fault.* A fault that is likely to manifest itself as a failure under the scope inputs subset.

Definition 6: *Out-of-scope fault.* A fault that is not likely to manifest itself as a failure under the scope inputs subset.

To make the above definitions meaningful, we need to provide a definition for "likely". With respect to the results reported in Chapter 4, we evaluated the likelihood of a given fault to manifest itself as a failure by using randomly generated test suites and mutation testing (this will be detailed in Section 4.3.2). For the studies reported in Chapter 6, on the other hand, we run all the test cases from the test pool created for our exploratory study over different *faulty* versions of our study subjects so to have the precise probability with which a given fault would be revealed.

Definition 7: *Traditional coverage.* The term "traditional coverage" is used to refer to the historical way of measuring coverage in which the number of entities expected to be covered embraces the full set of possible entities of a given kind (see Equation 1.1).

Definition 8: *Relative coverage.* The term "relative coverage" is used to refer to the way of measuring coverage in which the set of entities expected to be covered is customized according to the specific testing scope (see Equation 1.2).

1.4 Structure of the Thesis

In this section we outline the structure of the subsequent chapters.

Chapter 2 provides an overview of the main topics related to this thesis with some definitions and references. It aims at providing the reader with a short reference guide to the relevant literature that supported our work. It is not our intention in this chapter to cover all the related work as those will be provided along with each chapter that introduces a new adequacy criterion.

Chapter 3 depicts some testing scenarios that highlight the importance of the research conducted in this thesis. For each testing scenario illustrated, we discuss some of the challenges of carrying out software testing in that specific context. Then, in further chapters we introduce new approaches that could help to overcome the challenges presented.

Chapter 4 introduces a new coverage criterion tailored for the context of software reuse when the source code is available. In particular, we demonstrate how the newly introduced coverage criterion could be adopt for improving traditional test case prioritization, selection and minimization approaches.

Chapter 5 builds on top of Chapter 4 and complements it by proposing a new coverage criterion for the context of software reuse that could be adopted when the source code is not available.

Chapter 6 introduces the first of its kind coverage criterion for operational profile based testing and demonstrates how it could be adopted for test adequacy and test case selection. The proposed approach takes into account how frequently the program's entities are exercised to reflect the profile of usage into the measure of coverage.

Chapter 7 presents our conclusions and points out some directions for future work.

1.5 Published Material

The content of Chapter 4 has been partially published in:

[91] Breno Miranda and Antonia Bertolino. Improving test coverage measurement for reused software. In *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2015, Madeira, Portugal, August 26-28, 2015*, pages 27–34, 2015.

[93] Breno Miranda and Antonia Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, 2016.

Parts of Chapter 5 have been published in:

[90] Breno Miranda and Antonia Bertolino. Social coverage for customized test adequacy and selection criteria. In *9th International Workshop on Automation of Software Test, AST 2014*, pages 22–28, 2014.

[89] Breno Miranda. A proposal for revisiting coverage testing metrics. In *ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden - September 15 - 19, 2014*, pages 899–902, 2014.

Finally, the content of Chapter 6 has been partially published in:

[92] Breno Miranda and Antonia Bertolino. Does code coverage provide a good stopping rule for operational profile based testing? In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST 2016, pages 22–28, 2016.

2

Background

In the previous chapter we presented the context, objectives and contributions of this thesis along with some general definitions that will be frequently referenced throughout this document.

In this chapter we introduce the main topics related to this thesis with some definitions and references. The content provided here is not comprehensive and does not aim to cover all the related work as those are presented later for each chapter presenting a new adequacy criterion inspired by the idea of relative coverage.

Here we cover the following topics: coverage testing; prioritization, selection and minimization for regression testing; testing of reused code; software reliability and operational testing; and dynamic symbolic execution.

2.1 Coverage Testing

Goodenough and Gerhart [48, 49] made an early breakthrough in research on software testing when they introduced, in the 70's, the notion of a test criterion, i.e., the criterion that defines what constitutes an adequate test. Since then, objective measurement of test quality has been deeply explored by researchers. A lot of contributions have been made on the definition of new test criteria (e.g., [102, 135, 61, 37], just to mention a few). Considerable research effort has also been devoted to the comparison of multiple criteria to provide support for the use of one criterion or another [15, 130, 40, 132, 39].

A second breakthrough occurred in the late 80's when Frankl and Weyuker proposed to circumvent the problem of non-applicability of the data flow testing criteria by requiring the test data to exercise only the feasible entities [37]. For the first time it was possible to achieve 100% coverage, and this encouraged the proposal and evaluation of a multitude of adequacy criteria in the years to follow.

Still in the late 80's, many research works started to be developed with the objective of identifying the influence of increased test coverage on the fault detecting ability of different adequacy criteria [15, 30, 39, 37, 103, 132, 135]. After almost three decades, the debate on the topic does not seem to decline, and current testing research still seeks an answer to questions such as "Is branch coverage a good measure of testing effectiveness?" [129]. A recent experiment on large scale conducted by

Inozemtseva and Holmes [64] concludes that high coverage measures achieved by a test suite do not necessarily indicate that the latter also yields high effectiveness. This observation suggests that generating test cases for coverage as a target may be risky, as warned from many sides (e.g., [51, 84, 120]). A follow up work done by Zhang and Mesbah [139] concluded that the number of test assertions and assertion coverage are strongly correlated with a test suite's effectiveness. Thus, the argument continues.

Currently, the software testing literature contains two different but closely related notions associated with the term test adequacy criteria. First, an adequacy criterion is considered to be a *stopping rule* that determines whether sufficient testing has been done that it can be stopped. Second, an adequacy criterion provide *measurements of test quality* (e.g., the percentage of code coverage) when a degree of adequacy is associated with each test set so that it is not simply classified as good or bad. More formally, these two notions can be defined [141] as follows:

Definition 9: *Adequacy Criteria as Stopping Rule.* A test data adequacy criterion $\mathcal{C}$ is a function $\mathcal{C} : P \times S \times T \rightarrow \{true, false\}$. $\mathcal{C}(p, s, t) = true$ means that t is adequate for testing program p against specification s according to the criterion $\mathcal{C}$, otherwise t is inadequate.

Definition 10: *Adequacy Criteria as Measurement.* A test data adequacy criterion $\mathcal{C}$ is a function $\mathcal{C} : P \times S \times T \rightarrow [0, 1]$. $\mathcal{C}(p, s, t) = r$ means that the adequacy of testing the program p by the test set t with respect to the specification s is of degree r according to the criterion $\mathcal{C}$. The greater the real number r, the more adequate the testing.

An adequacy criterion can also be used as an explicit specification for test case selection. In such case, it is usually referred to as a *selection criteria*. With regards to categorization, adequacy criteria are typically classified according to the source of information used in the adequacy measurement and they can be:

- Specification-based: The required testing is defined based on the identified features of the software specification. A test set is considered adequate if all the identified features have been fully exercised.
- Program-based: Criteria are defined based on the program under test and a test set adequacy is evaluated according to weather the program has been thoroughly exercised.
- Combined specification-based and program-based: Incorporates ideas from both specification-based and program-based criteria.
- Interface-based: Adequacy criteria are defined based solely on the interface, that is, based on the types and valid ranges of inputs to the software without reference to any internal features of the specification or the program.

Another possible way of classifying adequacy criteria is according to their underlying approach to testing. There are three basic approaches:

- Structural testing: Establishes test requirements based on a particular set of elements in the program's structure or in the program's specification.
- Fault-based testing: The focus is on detecting faults in the software. An adequacy criterion from this approach is based in some measurement of the fault detecting ability of the test sets.
- Error-based testing: Test requirements are defined to check the program based on the knowledge of where errors typically occur.

In the following section we illustrate a few examples of test adequacy criteria.

Examples of Test Adequacy Criteria

The most common adequacy criteria in the literature are those based on a particular set of elements in the program's structure. Two main groups of program-based adequacy criteria exist: *control-flow* and *data-flow*. Most of the adequacy criteria from these two groups are based on the flow graph model of program structure. Shortly, a flow graph is a directed graph representing the paths that a program might traverse during its execution. The nodes in the flow graph represent basic blocks of statements, while the edges represent transfer of control between the basic blocks. Generally, each edge is associated with a predicate that represents the condition of control transfer between the nodes it connects.

Control-flow-based Criteria

Control-flow-based adequacy criteria aim at covering all the statements or blocks of statements, or specified combinations of them [63]. Some examples of control-flow-based adequacy criteria are given below:

- *Statement coverage*: It is the most basic adequacy criterion. It requires that all statements in the program under test (i.e., all the nodes in the program's flow graph) are exercised at least once.
- *Branch coverage*: It is a slightly stronger requirement when compared to statement coverage as it requires that all control transfers in the program under test (i.e., all the edges in the program's flow graph) are exercised at least once. Back in the 90's, Beizer [16] stated that statement coverage and branch coverage are the minimum mandatory testing requirement.
- *Path coverage*: Requires that all the execution paths from the program's entry to its exit (i.e., all distinct paths through the flow graph) are executed during testing.

Data-flow-based Criteria

The previously illustrated control-flow-based adequacy criteria are defined according to the basic flow graph. The data-flow-based adequacy criteria, on the other hand, are defined based on an augmented version of the flow graph that contains information about the data definitions and uses. Each variable occurrence is classified as either a definition occurrence or a use occurrence. Briefly, a definition occurrence happens when a value is assigned to a variable occurrence and a use occurrence happens when the value of a variable occurrence is referred. Use occurrence can be further classified as computational use (when a variable is used to compute a value for defining other variables or as an output value) or predicate use (when a variable is used for selecting execution paths after deciding whether a predicate is true). With these definitions, each node in the augmented flow graph can be marked as a definition or use node with respect to a particular variable. The most well-known data-flow adequacy criteria were proposed by Frankl and Weyuker [37]. Some examples are given below:

- *all-definitions*: Requires that all definition occurrences are covered (i.e., the set of execution paths through the augmented flow-graph must contain at least one path from each definition node to a use node that is covered by that definition).

- *all-uses*: It is a stronger requirement when compared to all-definitions. It requires that all of the uses should be exercised by testing (i.e., the set of execution paths through the augmented flow-graph must cover all use nodes of variables).

- *all-definition-use-paths* (or *all-DU-paths*, for short): Requires that all definitions of a variable and all paths through which that definition reaches a use of that variable are covered (i.e., the set of execution paths through the augmented flow-graph must cover all the paths between definitions and uses of variables).

Test Oracles

When conducting software testing, a fundamental assumption is that there exists some mechanism for determining whether the execution of a test case has passed or failed. Such a mechanism is called a *test oracle* [12, 66]. As anticipated in the introduction, achieving high levels of coverage is not sufficient to guarantee that appropriate testing has been accomplished. This statement is particularly influenced by the fact that test cases can exercise some entities without having proper oracles to guarantee that the expected behavior of the software under testing is satisfied. In other words, without strong and accurate test oracles, test cases are not helpful in revealing potential faults. This is in line with the work from Zhang and Mesbah [139]

that found that test suite's effectiveness is strongly correlated with the number of test assertions and assertion coverage.

Test oracles generation as well as their quality evaluation is out of the scope of this thesis. However, as we propose to focus the testing efforts on the entities that are relevant for a given testing context, one immediate impact that our work might have with regards to test oracles is that, by minimizing the number of entities to be covered, the expected number of test oracles required might also decrease. Naturally, the savings in test effort would be more pronounced in the cases where the test oracles need to be derived manually.

Test Coverage Tools

Test coverage tools provide objective measures about the extent to which a program has been exercised by a given test suite. They typically use code instrumentation, which is the insertion of additional code instructions to monitor specific components in a system, to identify the coverage items that have been executed. Generally, this kind of tool implements one or more adequacy criterion such as those previously illustrated, and they are able to provide information about which elements (e.g., statements, branches, variable definitions, and so on) were covered (and which ones were not) during a given test session. Knowing which elements have not been exercised is an important information for helping developers to enhance their test suites so to cover the untested elements.

The number of academic or commercial coverage tools are quite extensive to be cited completely. For the sake of brevity, we mention only a few examples: EMMA [1], JCov [2], and Cobertura [3] (for Java); Clover [4] (for Java and Groovy); OpenCover [5] (for .NET); NCover [6] (for C#.NET); gcov [7] (for C/C++); Devel::Cover [8] (for Perl); Coverage.py [9] (for Python); and many others.

In this thesis we claim that the traditional way of measuring coverage may not be suitable for several testing contexts because it is not the case that all the entities of a given program are of interest in all the contexts. To try to overcome this problem, we propose and evaluate a way of measuring coverage in which each entity of a given program is evaluated according to its relevance to the specific testing scope. In Chapters 4, 5, and 6 we present how the proposed approach could be applied to different testing scenarios. More precisely, in Chapter 4 we propose an adequacy

[1] `http://emma.sourceforge.net/`
[2] `http://yp-engineering.github.io/jcov/`
[3] `http://cobertura.github.io/cobertura/`
[4] `https://www.atlassian.com/software/clover/overview`
[5] `http://github.com/OpenCover/opencover`
[6] `http://ncover.sourceforge.net/`
[7] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
[8] `http://pjcj.net/perl.html`
[9] `http://www.ravenbrook.com/project/p4dti/master/test/coverage.py`

criterion customized for the context of software reuse. In Chapter 5, we introduce an adequacy criteria for the context of service-oriented architecture. Finally, in Chapter 6, we propose a stopping rule for operational profile based testing.

2.2 Prioritization, Selection and Minimization for Regression Testing

Regression testing aims at making sure that software that was previously developed and tested still behaves as expected after it was changed or interfaced with another software. Ideally, all the test cases that were created to verify the target software should be run again as changes in some part of the software may affect other parts as well. This approach may also be referred to as the *retest-all* strategy in the literature. However, because the test suites tend to grow in size as software evolves, the retest-all strategy may be impractical in most of the cases as the available resources (e.g., time, workforce) might not be sufficient to allow for the re-execution of all test cases.

To cope with this problem, many approaches have been devised to maximize the cost-effectiveness of regression testing when re-executing all test cases is not possible. Notable approaches are prioritization, selection and minimization. Although these approaches are not tied to regression testing, in the testing literature they have been mostly investigated for retesting a software code after modifications. In the following, we provide an overview of these approaches. For a comprehensive view of the adoption of prioritization, selection and minimization for regression testing, please refer to the survey conducted by Yoo and Harman [137].

Test case *prioritization* consists into reordering the test cases saved into a test suite so that potential faults can be detected as early as possible. Prioritization per se does not involve a reduction of the test suite size, but only a modification of its execution ordering. This does not directly imply a cost reduction, however brings two important advantages. If fault detection is anticipated, debugging can start earlier. Moreover, should testing activity be terminated prematurely (e.g. because of budget restrictions), then having executed the test cases in the prioritized order ideally ensures that the available time has been well spent to perform the most effective ones.

Test case *selection* deals with the problem of selecting a subset of test cases that will be used to test the software with respect to a given testing objective. The majority of the selection techniques are *modification-aware*, i.e., they seek to identify test cases that are relevant to some set of recent changes in the software under test (in such context, it may also be referred to as Regression Test case Selection, or RTS, for short). However, test case selection can also be oriented towards different testing objectives: it can focus, for example, on selecting tests to exercise the parts of the software that are too expensive to fix after launch (risk-based test case selection);

or it can focus on ensuring that the software is capable of completing the core operations it was designed to do (design-based test case selection) [73].

Test suite *minimization* is a technique that seeks to reduce as much as possible the test suite size by identifying and eliminating redundant test cases from it. The test suite minimization problem is equivalent to the well-known set covering problem which is NP-complete [71]. For this reason, the application of heuristics is encouraged.

In Chapter 4 *we propose to use scope information to improve test prioritization, selection and minimization for the purpose of testing reused code.* For the exploratory study reported in Chapter 4, we evaluated our approach applied to:

- Four prioritization techniques:
  - two coverage-based prioritization approaches: the well-known *total* and *additional* greedy heuristics;
  - one similarity-based approach proposed by Jiang et al. [69]; and
  - one search-based technique: an instantiation of the Hill Climbing algorithm applied to test prioritization as proposed by Li and coauthors [79].
- One selection heuristic based on the greedy additional algorithm;
- One minimization heuristic: the GE heuristic proposed by Chen and Lau [25].

2.3 Testing of Reused Code

Software reuse is the use of existing software or software knowledge to construct new software [36]. The basic insight behind it is that most software systems are variants of previous systems, rather than completely new systems. The purpose of software reuse is, then, to leverage this insight to improve software quality and productivity. As a field of study in software engineering, software reuse is often traced to Doug Mcilroy's paper from 1968 [85] which proposed the mass production of software components. Quoting Mcilroy:

> "*Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance. Existing sources of components lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors.*"

Since the early days of software engineering discipline, reuse has been advocated as a key solution to face the "software crisis" and improve software quality while

reducing time-to-market [85, 110, 111]. Moreover, integrating pre-existing components or portions of code that had already been widely tested and used seemed a prelude to easily achieve more reliable systems, as showed e.g. in [94]. However decades of research and practice in software reuse have not yet fully realized such optimistic expectations. Along with several success stories, the literature reports about many failures [96, 77], some also with catastrophic consequences as the often cited disasters of Therac-25 accidents [78] and of Ariane 5 Flight 501 [81], both eventually attributed to bad test reuse practices.

The literature indicates the existence of many software reuse strategies. Ravichandran and Rothenberger [113] highlighted three strategies that have received great attention from academia in the past years. Each of them, poses different challenges for the testing of the reused artifacts:

- *White-box reuse*: This strategy is characterized by the use of existing software artifacts found in in-house repository to build new software. It gives developers the freedom to modify the code to suit their needs, which provides a flexible way to harvest software assets in development projects by fitting existing components to new requirements. While code modification maximizes reuse opportunities this aspect is also the source of many problems as it requires developers to have a high degree of familiarity with the code.

- *Black-box reuse with in-house component development*: Similarly to white-box reuse, the focus is also on the use of existing software artifacts found in in-house repository to build new software. However, even though the source code is available, modification is not allowed. The reuse asset must be used "*as is*" and customization is possible only via configuration through defined parameters and switches.

- *Black-box reuse with components from markets*: This is analogous to the black-box reuse with in-house component. The main difference is that developers can look for potential reusable components not only in the in-house repositories, but also in component markets. This increases the chances of finding suitable components as the search space is broadened. The source code is, generally, not made available and, in any case, modification of code is not allowed. Customization is possible only via configuration through defined parameters and switches.

As Weyuker warned in [131], *developers reusing a component need to do considerable testing to ensure the software behaves properly in its new environment.* In this regard, the results from a recent survey among developers from different companies on the current practice of testing of reused software [124] are dismaying: to the question whether they tested a component in any way before reusing it, only 43% of the interviewed developers answered yes, while 41% did it rarely and 16% not at all. From the interviews it emerged that a reason at the origin of such risky

behavior was the lack of proper and simple-to-use tools, in particular for facilitating the testing of code that has been written by someone else. The authors report that many developers would like to see a tool that could give them some measure on how well they have tested a given function, class or code portion, as for example code coverage tools.

When the source code is available, as in the case of reuse of components from in-house repositories [113], measuring the test coverage of reused code would be useful for developers [124]. The problem is that when testing reused software, traditional ways to measure test coverage might produce meaningless information: in fact, not all entities might be of interest when a code is reused in a new context and environment.

Reaching 100% test coverage might not be a sensible target from within a certain scope in which some parts of a reused code would never be invoked. A more useful information for testers would be to know how much coverage they could achieve within their reuse scope. Thus, instead of measuring coverage as the ratio between the entities covered and the total number of entities, we suggest to measure coverage of reused code as the ratio between the entities covered and those that could potentially be exercised in the new scope.

In Chapter 4 we present an instantiation of the relative coverage criterion to the context of software reuse. More precisely, *we propose to use scope information to improve test prioritization, selection and minimization for the purpose of testing reused code.*

2.4 Software Reliability and Operational Testing

Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment [8]. Software reliability-engineered testing (SRET), in turn, is the combination of quantitative reliability objectives with operational profiles [99]. Operational profiles guide developers in testing more realistically, which makes it possible to track the reliability actually being achieved. Quoting Musa [100], "*a software-based product's reliability depends on just how a customer will use it. Making a good reliability estimate depends on testing the product as if it were in the field*".

Operational profile based testing is grounded on the notion that not all faults have the same importance. Depending on how it will be exercised by the users, a program can show quite different levels of reliability [82], because the probability that an existing fault becomes a failure may vary widely. Hence, as originally discussed in Adams' seminal study [2] and followed up by the now well-established software reliability discipline [82], optimized testing resources should focus on those functions that will be more frequently used.

The use of an operational profile to guide testing ensures that if the testing activitites are terminated and the software is delivered because of schedule constraints,

the most-used operations will have received more attention and the reliability level will be the maximum that is practically achievable for the given test time [100].

Authors and software testing practitioners define and apply operational profiles in ways that differ to various extents. For a comprehensive view of software testing supported by operational profile, we refer the reader to a recent survey conducted by Smidts and coauthors [119].

Concerning the relation between code coverage and software reliability, almost all studies assessing the effectiveness of coverage testing have used as a measure of effectiveness the faults (or mutations) detected without distinguishing their probability of failure in use (e.g., [64, 75, 120, 129, 134], just to cite a few). Thus, we still know little about how coverage testing is related or not to *delivered reliability* [38]. An exception is the study by Del Frate and coauthors [27], who observed that the relation between coverage and reliability varied widely with subject's size and complexity. Moreover, all the previously cited studies considered traditional coverage measures, which require that *all* entities are covered *at least once*. In other words, all entities are considered as having the same relevance for the purpose of completing coverage. This assumption seems contradictory with the very idea of reliability according to which the program's functions should be exercised more or less frequently accordingly to user profiles, and in doing this, code entities as well will consequently be exercised with different frequencies.

In Chapter 6 we will introduce a coverage criterion for operational profile based testing. Our work is the first attempt to bridge the notion of operational testing with coverage measure. More precisely, we propose an approach that takes into account how much the program's entities are exercised, i.e., we distinguish between entities that are very often exercised from those that are scarcely exercised to reflect the profile of usage into the measure of coverage.

2.5 Dynamic Symbolic Execution

Symbolic execution is a way of analyzing a program to determine which inputs cause each part of the program to execute. It was originally proposed in the 70's [74] and thanks to the combination of increased availability of computational power with advances in constraint solving technology, nowadays it has become a hot topic in the software engineering research.

Symbolic execution has been actively researched for different application domains. It has been applied to *intrusion detection/prevention systems* to find vulnerability signatures and prevent attacks that target specific vulnerabilities [18, 19]. It has also been applied to *malware analysis* with the objective of analyzing how information flows through a malware [97, 17]. Besides that, today there are several tools available that perform a hybrid of symbolic and concrete execution, in conjunction with constraint solving techniques, so to automatically generate a set of test cases that target code coverage. Such state-of-the-art approach for *auto-*

mated test generation, named Dynamic Symbolic Execution (DSE), has received a lot of attention from researchers [117, 45, 21, 109, 24]. Many of the most popular symbolic execution tools are open-source: Symbolic Java™PathFinder[10], jCUTE[11], CREST[12], and KLEE[13], just to mention a few. Symbolic execution tools have also been adopted by companies (e.g., Pex[14] and SAGE [46] are used at Microsoft) and government agencies (e.g., Symbolic Java™PathFinder is used by NASA).

While exploring different program paths for test case generation, most of the symbolic execution tools will also check for the presence of various kinds of errors including assertion violations, uncaught exceptions, security vulnerabilities, and memory corruption [22]. This ability to generate concrete test inputs is one of the most appreciated strengths of symbolic execution. From the test generation perspective, it supports the task of generating test suites that achieve high code coverage. From the bug-finding perspective, on the other hand, it can help developers by providing a concrete input that triggers a given bug. Such input can be used to confirm the failure regardless of the symbolic execution tool that generated it.

Despite the unquestionable benefits of using DSE, many challenges still need to be addressed so that more tools are developed outside the academic context. The *path explosion*, for example, is considered to be one of the key challenges faced by symbolic execution. Because the number of program paths is usually exponential in the number of static branches in the code, the exploration of all feasible paths may be prohibitively expensive. Thus, given a fixed time budget, it is critical to explore the most relevant paths first. To overcome this challenge, the main mechanism used by symbolic execution tools is to prioritize path exploration by using search heuristics (to focus on achieving high statement and branch coverage, for example). Other challenges include *floating-point computations*, *memory modeling* and *concurrency handling* [22].

In Chapter 4 we use KLEE [20] — a well-known symbolic execution tool capable of automatically generating tests that achieve high coverage even for complex and environmentally-intensive programs — to identify the set of in-scope entities to be used for computing relative coverage in the context of software reuse.

2.6 Search-Based Test Data Generation

In addition to dynamic symbolic execution, search-based test data generation is another widely used technique for generating test data. It has received a lot of attention from researchers in recent years [87, 55, 4, 54, 88] and it has been applied in multiple areas [86], including: the coverage of specific program structures, as part

[10] `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc`
[11] `http://osl.cs.illinois.edu/software/jcute/`
[12] `http://github.com/jburnim/crest`
[13] `http://klee.github.io/`
[14] `http://research.microsoft.com/en-us/projects/pex/`

of white-box testing strategy; the evaluation of specific program features, as part of black-box testing strategy; and the verification of non-functional properties, such as the worst-case execution time.

Different metaheuristic techniques have been adopted for test data generation. Among the most common ones are: Hill Climbing, Evolutionary Algorithms, and Simulated Annealing. For more details about the different metaheuristics and their application to the test data generation problem we refer the reader to the comprehensive survey from McMinn [86].

In Chapter 4 we illustrate the adoption of our scope-aided testing approach to support multiple prioritization techniques, including a search-based one. In particular, we show how scope information could be used by the Hill Climbing algorithm applied to test prioritization.

3

Motivation

In the previous chapter we presented a brief overview of the relevant literature that supported the work conducted in this thesis.

In this chapter, we present three testing scenarios that further emphasize the importance of the research conducted in this thesis. First, we discuss the issues related to the testing of code that is reused in a context that is different from the original one in which they were conceived. Then, we illustrate the scenario of a service provider that is (re)used by developers that integrate their own services with those from the provider. Finally, we depict a scenario of a system that is used by a heterogeneous customer base.

In each subsection, we first illustrate the testing scenario through a simple example and, then, we briefly discuss some of the challenges of carrying out software testing in that specific scenario.

3.1 Testing of Reused Code in a New Context

When a program uses reused code or third-party components in a context that is different from the original one, some of their entities (e.g. branches) might never be exercised. Consider, for example, the small program displayed in the leftmost column of Table 3.1 which contains two functions, `Example1` and `Example2`.

As highlighted in Table 3.1, we assume two faulty statements in the function `Example1`. The first one can only be triggered if Statement 2 is executed: hence only test cases with `x` bigger than 0 are likely to reveal this fault. The second fault, on the other hand, can only be triggered when Statement 5 is traversed: hence only test cases with `x` equal to or less than 0 are likely to expose this fault.

Assuming a statement-coverage approach, Table 3.1 also provides information regarding a set of test cases $T = [\mathrm{TC}_1,\mathrm{TC}_2,\mathrm{TC}_3,\mathrm{TC}_4,\mathrm{TC}_5]$ used to exercise the example code. Each test case is displayed along with the values used for the input variables (`x`, `y`, and `z`) as well as the statements covered (please ignore for the moment the last two lines, whose meaning is explained later on).

Let us now assume that the aforementioned code is going to be reused in a scope in which we know that `x` will *always* be bigger than 0 and the function `Example2` is no longer used. In Table 3.1, the statements exercised in such context are highlighted

Table 3.1: Statement coverage achieved by the test cases used to exercise the sample code

Code	TC1 (x=1,y=1)	TC2 (x=-1,y=-1)	TC3 (x=1,y=-1)	TC4 (z=0)	TC5 (z=10)
Example1(x, y):					
1. if x >0:	✓	✓	✓		
2. s2 //fault 1	✓		✓		
else:					
3. s3		✓			
4. s4		✓			
5. s5 //fault 2		✓			
6. if y >0:	✓	✓	✓		
7. s7	✓				
else:					
8. s8		✓	✓		
9. s9		✓	✓		
Example2(z):					
10.if z == 0:				✓	✓
11. s11				✓	
else:					
12. s12					✓
#st. covered	4	7	5	2	2
#in-scope st.	4	4	5	0	0
#out-of-scope st.	0	3	0	2	2

in grey: we see that Statement 5 would never be reached and so *fault 2* would never be triggered. In our terminology, *fault 2* is said to be out-of-scope, whereas *fault 1* that can be triggered under the known constraint is an in-scope fault.

In Chapter 4 we discuss how the reuse context information could be leveraged to improve test prioritization, selection and minimization. In particular, the proposed approach takes into account the possible constraints delimiting the new input domain scope and aims at detecting those faults that under such constraints would be more likely triggered.

3.2 Testing in the Absence of Code Coverage Metrics

Service Oriented Architecture (SOA) is a very attractive architectural pattern as it allows pieces of software developed by independent organizations to be dynamically

composed to provide richer functionality [14]. However, the same reasons that enable flexible compositions also prevent the application of some traditional testing approaches. Web services usually expose just an interface, which is enough to invoke the service's operations and to develop some general (black-box) test cases. Such an interface, however, is not sufficient for testers willing to evaluate the integration quality between their applications and the independent web services.

Let us consider the example of a service provider S implementing a Travel Reservation System (TRS). As illustrated in Figure 3.1, TRS includes several operations, relative to flight and hotel booking, car reservation, user registration, login and payment. These operations (in total 42 for our illustrative example) are made available through the service public interface. Many operations can be associated with a same group (e.g., the flight booking group contain the operations `FlightLookup`, `CheckSeatAvailability`, `getFlightTicketPrice`, and so on).

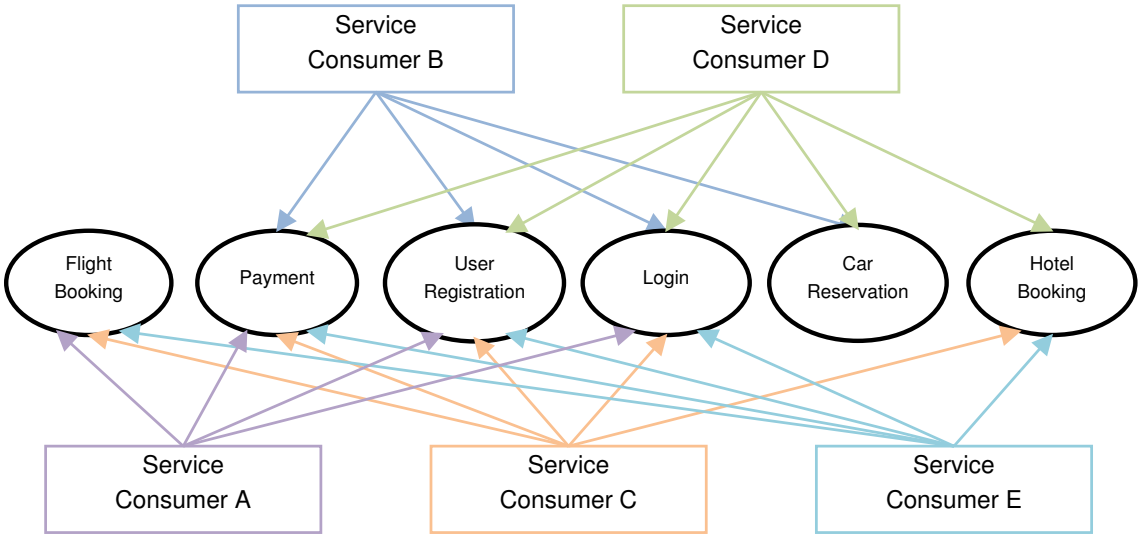


Figure 3.1: Travel Reservation System implemented by service provider S.

TRS services can be used by many consumers. In the example in Figure 3.1 we show five different service consumers, each using different combinations of the provided operations. Service consumer A, for example, uses operations from the flight booking, payment, user registration, and login groups, suggesting that it is a service being used in a travel agency. Service consumer B, on the other hand, adopts operations from car reservation, user registration, login, and payment, suggesting that the service is being used by a car rental company. Service consumers A and B are different clients making use of the same service provider.

Let us now assume that a new service, say service consumer N, is developed. It is used in a travel agency and, as depicted in Figure 3.2, it is implemented to invoke the operations available from: flight booking, hotel booking, user registration, login, and payment. To diligently test the integration of the new service N with TRS, a

tester creates a test suite to exercise the operations implemented by the new service N as well as the operations used from the service provider S.

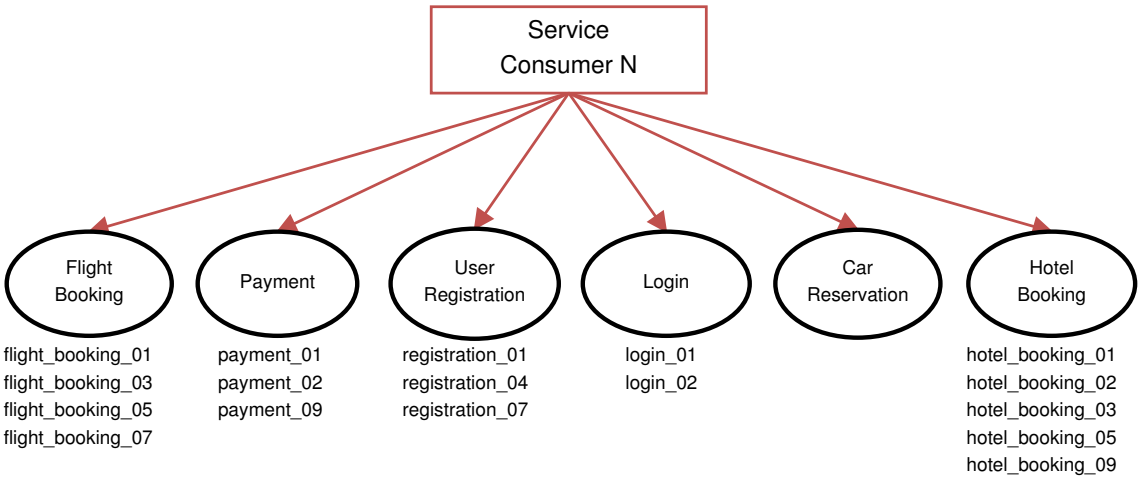


Figure 3.2: Service consumer N.

Let us assume that the test suite covers a total of 17 operations (those listed in Figure 3.2), and the tester is interested in assessing coverage over S (in terms of operation coverage). Traditional coverage, as seen in Equation 1.1, would be calculated by dividing the amount of operations invoked by N by the total amount of operations available in the service provider S.

Such a coverage metric would be helpful if the tester were interested in thoroughly testing the service provider S considering *all* the operations, including the ones that are never used by N, such as those in the car reservation feature. If, on the other hand, the tester is deliberately not interested in some operations, which, in fact, is the case in this example, this coverage information would not be meaningful.

Even worse, let us suppose that, besides calculating the ratio of covered operations, service N would also like to get the list of un-tested operations among those provided by S. Again, this list would be of little value if it cannot help testers to identify possibly relevant operations that have been left un-tested. In this very simple example, using the traditional approach to measure coverage, such list would contain among un-tested operations also those from car reservation; for really large applications, the number of not relevant operations could be so big that providing such information would be useless.

For the testing of N, we would like to measure coverage over the actually relevant operations, and not over the whole set of 42 operations. The approach we present in Chapter 5 assumes that in an effort to better support customers in their testing activities, service provider S is willing to provide relevant coverage information to them.

3.3 Testing of Systems with Heterogeneous Customer Base

Let us consider the example of a publication management system used by different types of users. There are the *authors*, who use the system mainly to add new publications and to browse the existing ones. The *librarians*, who make sure that all the publications added to the database meet the library guidelines. Librarians are also in charge of making weekly backups to make sure that the database containing the publications of their respective institutes is safe. And there are the *system administrators*, who are in charge of adding and removing users, granting permissions according to the user role, and making daily backups of the whole system to make sure that all the publications are safe.

After the last update, the publication management system started presenting intermittent issues during the backup operation. These issues go unnoticed by the authors as they do not make use of the backup operation. This group of users feels that the system is reliable as it effectively fulfills the users' needs. The librarians notice occasional problems while performing the backup. For them, the system is fairly reliable as it meets their needs *most of the time*. The system administrators, on the other hand, will notice the failures in the backup operation very frequently as they need to perform this action in a daily basis. Their opinion regarding the system's reliability is certainly very different from the other groups of users.

This is a classic scenario of a system with heterogeneous customer base. In Chapter 6 we investigate (i) how one could decide when to stop testing and (ii) how test cases could be selected if the testing activities are carried out with a specific group of customers in mind.

3.4 Concluding Remarks

The three examples illustrated in this chapter are just a few of the many possible scenarios for which the application of traditional coverage metrics may not be suitable. In the next three chapters, we propose different instantiations of the relative coverage idea applied to each of the above scenarios. More precisely, in Chapter 4 we propose an instantiation of the relative coverage for the context of software reuse and we show how it could be adopted for improving traditional approaches for test prioritization, selection and minimization. In Chapter 5, we discuss how the relative coverage approach could be adopted in the context of service-oriented architecture. Finally, in Chapter 6, we argue that the concept of relative coverage could also be useful as a stopping rule for operational profile based testing.

4

Redefining Coverage Criteria for the Context of Software Reuse

In the previous chapter we presented a few examples to illustrate that the traditional way of calculating coverage may not be suitable for every testing scenario.

In this chapter, we introduce a new coverage criterion that tries to overcome the difficulties illustrated in the motivating example 3.1. It is called "Relevant Coverage" and, in each testing context in which a code is (re)used, it calculates coverage measures over the set of relevant entities for that context. Here we present how the relevant coverage could be adopted to boost traditional approaches for test case prioritization, selection and minimization .

We start by describing our approach to calculate relevant coverage in Section 4.1. In Section 4.2 we discuss how the newly introduced coverage criterion could be used to boost traditional approaches for test case prioritization, selection and minimization. We then describe the settings of our empirical evaluations in Section 4.3, including the Research Questions, the study subjects, tasks, procedures and metrics. Results are reported and illustrated separately for prioritization, selection and minimization in Sections 4.4, 4.5, and 4.6, respectively. We discuss costs and benefits of the approach in Section 4.7, and threats to the validity of our study in Section 4.8. Preliminary results of an ongoing investigation on the adoption of relevant coverage for test adequacy are reported in Section 4.9. Related Work is discussed in Section 4.10 and, finally, our concluding remarks are presented in Section 4.11.

4.1 Relevant Coverage

The rationale behind relevant coverage is that in each scope in which a given system is being (re)used, the in-scope entities are those that could be potentially exercised according to the specific input domain restrictions.

In the motivating example 3.1 it was stated that the code from Table 3.1 would be reused in a context where it was known that x would be *always* bigger than 0 and that the function `Example2` would no longer be used. With these input domain restrictions, when considering statement coverage for such scenario, the set of in-scope entities are the statements highlighted in grey in Table 3.1.

This coverage metric is analogous to the one for calculating coverage in the traditional way and it is defined in Equation 4.1.

$$\textbf{Relevant coverage} = \frac{number\ of\ covered\ entities}{number\ of\ entities\ in\ the\ reuse\ scope} \cdot 100(\%) \qquad (4.1)$$

As it can be noticed from Equation 4.1, for computing relevant coverage first we need to identify the set of entities in the *reuse scope*, i.e., the set of in-scope entities. This is achieved by the following steps:

1) Constraints identification:

As the first step of our approach, we use the information regarding the specific testing scope to identify the input domain constraints that will be further used to identify the in-scope entities. The input domain constraints provided could vary, they could be coarse grained such as a list of functions that are expected to be used in that specific scope; or fine grained such as the precise range of values expected to be used by a given variable. The more information is provided, the more precise is the identification of the in-scope entities.

In the example from the motivating scenario 3.1, the constraints provided were the two facts that, in the new test context, (i) variable x would always be bigger than 0; and (ii) function `Example2` would be no longer used. Other constraints such as the range of values expected for the variable y, for example, could also be provided.

2) In-scope entities identification:

Different approaches could be adopted to identify the entities that are relevant under the input domain constraints collected in the first step of our approach. For example, one could apply a reachability algorithm on the static call graph of the given program. Even though this is an undecidable problem [98], there exist algorithms capable of generating approximated solutions.

We decided to use Dynamic Symbolic Execution (DSE) since it has shown to be a powerful approach to analyze the code dynamically and guide its exploration based on the input domain constraints [109, 24]. Our decision was influenced by the fact that DSE is very actively investigated and several tools are available. For the exploratory studies reported in this chapter, we adopted KLEE [20], a well-known symbolic execution tool capable of automatically generating tests that achieve high coverage even for complex and environmentally-intensive programs.

To guide the DSE exploration, we provide the input domain constraints (collected in the previous step of our approach) to KLEE. When it is run on the target code, KLEE tries to explore all paths, finding concrete test inputs to exercise them. The set of in-scope entities consists of those exercised by the DSE-generated test cases[1].

[1]Note that we "replay" the test cases generated using the original program and use gcov so to get accurate coverage achieved by the DSE-generated test cases.

Indeed, if some entities are not exercised by the DSE-generated test cases it is because they are not reachable under the input domain constraints provided[2].

4.2 Prioritization, Selection and Minimization

In this section we discuss how the newly introduced coverage criterion, the relevant coverage, could be used to boost traditional approaches for test case prioritization, selection, and minimization. To illustrate our approach we will refer to the example displayed in Table 4.1. This table is the same one shown in the motivating scenario 3.1 in Chapter 3. We show it again in order to provide additional details that are relevant for the explanation of our approach.

Table 4.1: Statement coverage achieved by the test cases used to exercise the sample code

Code	**TC1** (x=1,y=1)	**TC2** (x=-1,y=-1)	**TC3** (x=1,y=-1)	**TC4** (z=0)	**TC5** (z=10)
Example1(x, y):					
1. if x >0:	✓	✓	✓		
2. s2 //fault 1	✓		✓		
else:					
3. s3		✓			
4. s4		✓			
5. s5 //fault 2		✓			
6. if y >0:	✓	✓	✓		
7. s7	✓				
else:					
8. s8		✓	✓		
9. s9		✓	✓		
Example2(z):					
10.if z == 0:				✓	✓
11. s11				✓	
else:					
12. s12					✓
#st. covered	4	7	5	2	2
#in-scope st.	4	4	5	0	0
#out-of-scope st.	0	3	0	2	2

[2]This is true under the assumption that the dynamic symbolic exploration will have enough time to fully explore the target code and that precise solutions will be found for all the constraints detected. In the cases where, for example, the DSE exploration is halted, or it encounters non-linear constraints or black-box functions, this statement does not hold anymore.

Traditional white-box approaches for test case prioritization, selection, and minimization evaluate test cases based on how the entities in the program flowgraph are covered. For example, when applied to the example from Table 4.1 the adoption of the *additional* prioritization strategy would yield a prioritized test suite TS_p = [TC_2,TC_1,TC_4,TC_5,TC_3]. A test case selection heuristic based on the *additional* greedy algorithm would produce the test suite TS_s = [TC_2,TC_1,TC_4,TC_5]; And the test minimization approach as proposed by [25] would provide a minimized test suite TS_m = [TC_2,TC_1,TC_4,TC_5].

Our approach enhances existing traditional techniques by taking into account the relevance of the entities to be covered with respect to possible constraints delimiting the input domain scope. In the motivating scenario 3.1 it was assumed that the code from Table 4.1 would be reused in a scope in which we knew that `x` would always be bigger than 0 and the function `Example2` would no longer be used. Recall that the statements highlighted in grey are the ones exercised in such a new context. Our approach aims at exploiting this kind of context-related information to bias test case prioritization, selection, and minimization so to give priority to those test cases that are more likely to reveal in-scope faults.

Throughout the remainder of this chapter, we use the term *scope-aided prioritization* to refer to traditional prioritization techniques supported by our approach. Analogously, we use the terms *scope-aided selection* and *scope-aided minimization* to refer to selection and minimization techniques assisted by our approach, respectively. Finally, we use the term *scope-aided testing* to refer to these techniques collectively.

Scope-aided testing is not an approach per se, but a boost to existing prioritization, selection, and minimization approaches. For example, the application of scope-aided prioritization, when used to boost the *additional* approach, would provide a prioritized test suite $TS_{p'}$ = [TC_3,TC_1,TC_2,TC_4,TC_5]. Any of the prioritized test suites (either scope-aided or not scope-aided) would trigger the two faults, even though *fault 2* is very unlikely to manifest itself within the context scope in which `x` is always bigger than 0. However, the adoption of a scope-aided prioritized test suite brings the benefit of revealing *fault 1*, the critical one, faster.

The adoption of scope-aided selection to improve the selection heuristic would provide the test suite $TS_{s'}$ = [TC_3,TC_1], whereas its adoption to enhance the minimization approach proposed by [25] would yield the test suite $TS_{m'}$ = [TC_1,TC_2]. In both cases the scope-aided approach would preserve the fault detection capability of the original test suites (when considering in-scope faults) while bringing the benefit of yielding smaller test suites.

Table 4.2 provides a summary of the comparison between the test suites derived by traditional and scope-aided approaches. In Table 4.2, "*Prio.*", "*Sel.*", and "*Min.*" stand for prioritization, selection, and minimization, respectively.

The set of in-scope entities is used as input to our approach. We identify this set by following the steps described in 4.1. In Sections 4.2.1, 4.2.2, and 4.2.3, we detail how they are used for scope-aided prioritization, selection, and minimization, respectively. In each section, we first define the problem we are addressing and then

Table 4.2: Test suites derived by traditional and scope-aided approaches

	Traditional	Scope-aided	Note
Prio.	TS_p=[TC_2,TC_1,TC_4,TC_5,TC_3]	$TS_{p'}$=[TC_3,TC_1,TC_2,TC_4,TC_5]	Scope-aided test suite reveals the critical fault (*fault 1*) faster
Sel.	TS_s=[TC_2,TC_1,TC_4,TC_5]	$TS_{s'}$=[TC_3,TC_1]	Our approach preserves the fault detection capability (when considering in-scope faults) while yielding a smaller test suite
Min.	TS_m=[TC_2,TC_1,TC_4,TC_5]	$TS_{m'}$=[TC_1,TC_2]	Our approach preserves the fault detection capability (when considering in-scope faults) while yielding a smaller test suite

illustrate an instantiation of our scope-aided approach to support the problem at hand (using the motivating scenario depicted in Section 3.1). For each example, we first review how a considered traditional technique works, and then explain how the scope-aided approach could be adopted to improve testing of reused software.

4.2.1 Scope-aided Prioritization

Test case prioritization consists into reordering the test cases saved into a test suite so that potential faults can be detected as early as possible. Prioritization per se does not involve a reduction of the test suite size, but only a modification of its execution ordering. This does not directly imply a cost reduction, however brings two important advantages. If fault detection is anticipated, debugging can start earlier. Moreover, should testing activity be terminated prematurely (e.g. because of budget restrictions), then having executed the test cases in the prioritized order ideally ensures that the available time has been well spent to perform the most effective ones.

4.2.1.1 Definition of scope-aided prioritization

More formally, the test case prioritization problem is commonly defined [114] as follows:

Definition 11. (*Prioritization Problem*):

Given: *A test suite T; the set PT of permutations of T; a function f from PT to the real numbers $\mathcal{R}$*

Problem: *Find $T' \in PT$ such that $\forall T''$: ($T'' \in PT$) and ($T'' \neq T'$): $[f(T') \geq f(T'')]$*

In the above definition f is a given function that assigns some award value to a test suite order. Ideally, such function would refer to fault detection rate: the earlier the faults are detected, the more preferable a test suite order is. Unfortunately in reality we cannot know in advance which test case detects which fault, so prioritization approaches can only be based on surrogate criteria [137] that are demonstrated to give good approximations for fault detection effectiveness. Both black-box and white-box surrogate criteria have been devised. In this work, we focus on white-box prioritization approaches in which the ordering is done according to how the entities in the program flow graph are covered. Several different methods have been introduced providing different instantiations for the "how" in the sentence above: in additional greedy ordering [115] a test case t_i will precede another test case t_j if t_i covers more yet uncovered entities than t_j; in similarity-based prioritization [69] instead, t_i will go before t_j if t_i covers a flow graph path that is more dissimilar from all covered paths so far than t_j covered path; in search-based prioritization [79] t_i is preferred over t_j if t_i provides a higher increase for the adopted fitness function than t_j does.

We now introduce the notion of Scope-aided Prioritization in analogous way to that of Prioritization as follows:

Definition 12. (*Scope-aided Prioritization Problem*):

***Given**: A test suite T; the set PT of permutations of T; a testing scope $\mathcal{S}$; a function f_s from PT and $\mathcal{S}$ to the real numbers $\mathcal{R}$*

***Problem**: Find $T' \in PT$ such that $\forall T''$: ($T'' \in PT$) and ($T'' \neq T'$): [$f_s(T') \geq f_s(T'')$]*

Thus the problem of scope-aided prioritization is the same as prioritization, but we use an objective function f_s that awards those orderings that would reveal in-scope faults faster (more precisely, we use coverage of in-scope entities as a surrogate criterion for detection of in-scope faults).

The scope-aided prioritization approach can be adopted to boost a wide spectrum of traditional test case prioritization techniques. For the experiments reported in the following, we considered two coverage-based approaches (*total* and *additional*), one similarity-based approach, and one search-based approach.

4.2.1.2 The scope-aided approach applied to a prioritization example

In this section we illustrate our approach being applied to boost a coverage-based prioritization technique. Both *additional* and *total* strategies have been proposed [52]. For illustrative purposes, next we instantiate our approach on the *additional* strategy (application to total would be quite similar).

Algorithm 1 depicts the additional strategy applied for test case prioritization. On each iteration, the test case that yields the highest coverage is selected (line

3) and the coverage information of the remaining test cases is updated (line 6) to reflect their coverage with respect to the not yet covered entities.

Algorithm 1: Prioritization (greedy additional heuristic)

```
   Input: T                                   /*the test suite to be prioritized*/
          coverageInfo    /*list of entities covered by each test case from T*/
   Output: T'                                          /*a permutation of T*/
1  T' ⟵ [ ]                              /*T' is initialized as an empty list*/
2  while thereAreTestCasesToBePrioritized(T, T', coverageInfo) do
3  |  selectedTestCase ⟵ getNextTestCase(T, coverageInfo)   /*selects the test
   |   case that yields the greatest coverage*/
4  |  add(selectedTestCase, T')
5  |  remove(selectedTestCase, T)
6  |  updateCoverageInformation(coverageInfo, selectedTestCase)      /*adjusts the
   |   coverage information of the remaining test cases to indicate their
   |   coverage of entities not yet covered*/
   end
```

When multiple test cases cover the same number of not yet covered entities, an additional rule is needed to decide which one of these test cases will be selected. One possibility would be to simply pick one of the tied test cases randomly, but this would imply having a non-deterministic output. Our implementation of *getNextTestCase* sorts the tied test cases in ascending order according to their IDs in order to guarantee a deterministic output.

After prioritizing a subset of the test cases it is possible to reach a point in which all the entities are covered and the remaining test cases cannot contribute to increase coverage anymore. At this point, different approaches could be adopted to order the remaining test cases. One possibility could be resetting the coverage vectors of the remaining test cases and reapplying the additional strategy algorithm as done in [114]; another possibility could be adopting a different approach (e.g., the total strategy) to prioritize the remaining tests. Algorithm 1 does not adopt alternative strategies and keeps selecting the test cases in the same way, which basically means that when the maximum coverage is achieved, the remaining test cases are ordered according to their IDs.

When applied to the motivating example in Table 4.1, the first test case that Algorithm 1 selects — the one that achieves the highest coverage — is TC_2 that covers seven (1, 3, 4, 5, 6, 8, and 9) out of 12 possible statements. Then, it looks for the next test case that achieves the highest coverage with respect to the yet to be covered statements (2, 7, 10, 11, and 12). For the next choice, three test cases are tied (TC_1, TC_4, and TC_5) covering two statements each and TC_1 is the one selected (as explained before). The list of uncovered statements is then updated (10, 11, and 12). The next test case chosen is TC_4 and then TC_5. At this point, 100% of

the statements have been covered already and the remaining test cases are ordered according to their IDs. The prioritized test suite returned by Algorithm 1 would be TS_p = [TC_2,TC_1,TC_4,TC_5,TC_3].

Algorithm 2: Scope-aided Prioritization (greedy additional heuristic)

```
Input: T                                   /*the test suite to be prioritized*/
       coverageInfo     /*list of entities covered by each test case from T*/
       inscopeEntities     /*list of entities relevant to the current scope*/
Output: T'                                          /*a permutation of T*/
1 T' ⟵ [ ]                              /*T' is initialized as an empty list*/
2 while thereAreTestCasesToBePrioritized(T, T', coverageInfo, inscopeEntities) do
3 |   selectedTestCase ⟵ getNextTestCase(T, coverageInfo, inscopeEntities)
  |    /*selects the test case that yields the greatest coverage of in-scope
  |    entities*/
4 |   add(selectedTestCase, T')
5 |   remove(selectedTestCase, T)
6 |   updateCoverageInformation(coverageInfo, inscopeEntities, selectedTestCase)
  |    /*updates the set of in-scope entities and adjusts the coverage
  |    information of the remaining test cases to indicate their coverage of
  |    entities not yet covered*/
end
```

Scope-aided prioritization adopting the additional strategy is depicted in Algorithm 2. It is analogous to Algorithm 1 with the following modifications:

1. The set of in-scope entities, received as input, is used by *getNextTestCase* to decide which test case achieves the highest coverage;

2. If multiple test cases cover the same number of not yet covered in-scope entities, *getNextTestCase* tries to solve the tie by returning the test case that achieves the highest additional coverage when considering all the entities. If the tie persists, then the tied test cases are sorted in ascending order according to their IDs.

Modification 2 brings the advantage of making Algorithm 2 revert to the traditional prioritization approach (Algorithm 1) after all the in-scope entities are covered.

The second last line of Table 4.1 counts the number of statements covered by each test case considering only the ones that belong to the set of in-scope statements. When Algorithm 2 is applied, the first test case selected is TC_3 as it is the one that covers the highest number of in-scope statements. After TC_3 is selected, there is only one in-scope statement that needs to be covered (statement 7). TC_1 is then selected as it is the only test case that covers that statement. After TC_1 is selected, 100% coverage is achieved for the in-scope statements and the remaining test cases

are ordered as explained before. The test suite prioritized by Algorithm 2 is then provided as: $TS_{p'} = [\mathrm{TC}_3,\mathrm{TC}_1,\mathrm{TC}_2,\mathrm{TC}_4,\mathrm{TC}_5]$.

4.2.2 Scope-aided Selection

Test case selection deals with the problem of selecting a subset of test cases that will be used to test the software with respect to a given testing objective. The majority of the selection techniques are *modification-aware*, i.e., they seek to identify test cases that are relevant to some set of recent changes in the software under test (in such context, it may also be referred to as Regression Test case Selection).

However, test case selection can also be oriented towards different testing objectives: it can focus, for example, on selecting tests to exercise the parts of the software that are too expensive to fix after launch (risk-based test case selection); or it can focus on ensuring that the software is capable of completing the core operations it was designed to do (design-based test case selection) [73].

4.2.2.1 Definition of scope-aided selection

More formally, the problem of test case selection can be defined as follows:

Definition 13. (*Selection Problem*):

***Given**: A program P; and a test suite T*

***Problem**: Find a subset of T, T', such that testing P with T' preserves some desired property of testing P with T*

In the above definition, a desired property to be preserved, for example, could be covering a same set of entities; or covering the entities affected by changes in P with respect to a previous version.

The problem of scope-aided selection is analogous to the selection problem and it aims at selecting a set of test cases to test a given program with respect to a testing scope. It is defined in analogous way to that of selection as follows:

Definition 14. (*Scope-aided Selection Problem*):

***Given**: A program P; a test suite T; and a testing scope $\mathcal{S}$*

***Problem**: Find a subset of T, T', such that $T' \in \mathcal{S}$ and testing P with T' preserves some desired property of testing P with T*

In this work, because we focus on test case selection for reused software, which does not necessarily imply regression test selection, we use, for our instantiation example and for our exploratory study, a test selection technique that is not modification-aware. Yet our approach can also be applied to the regression test case selection problem.

4.2.2.2 The scope-aided approach applied to a selection example

To instantiate the scope-aided approach being applied to support a test case selection example, we refer to the following selection heuristic: use the greedy additional algorithm to repeatedly select the test case that covers the maximum number of uncovered entities until all entities are covered. This heuristic is depicted in Algorithm 3.

Algorithm 3: Selection (greedy additional heuristic)

```
   Input: T            /*the test suite from which test cases can be selected*/
          entities                              /*list of entities to be covered*/
          coverageInfo          /*list of entities covered by each test from T*/
   Output: T'          /*a subset of T, with which to test the target program*/
1  T' ⟵ [ ]                                  /*T' is initialized as an empty list*/
2  while thereAreUncoveredEntities(T, entities, coverageInfo) do
3  |  selectedTestCase ⟵ getNextTestCase(T, entities, coverageInfo)   /*selects
   |   the test case that covers the highest number of uncovered entities*/
4  |  add(selectedTestCase, T')
5  |  updateUncoveredEntities(entities, selectedTestCase)   /*removes the entities
   |   covered by the selected test case from the list of uncovered entities*/
   end
```

In Algorithm 3, the function *getNextTestCase* (line 3) behaves in the same way as that of Algorithm 1, that is, when multiple test cases cover the same number of not yet covered entities, it sorts the tied test cases in ascending order according to their IDs in order to guarantee a deterministic output.

Without the support of our approach, the first test case selected by Algorithm 3 is TC_2 as it covers the highest number of uncovered statements (7 out of 12). For the next choice, three test cases are tied (TC_1, TC_4, and TC_5) covering two statements each, and TC_1 is selected. After TC_1 is selected, three statements still need to be covered (10, 11, and 12) and TC_4 and TC_5 are tied again. After the tie is solved, TC_4 is chosen and then TC_5. When TC_5 is selected, 100% of the statements have been covered and test case selection stops producing the final test suite $TS_s = [TC_2, TC_1, TC_4, TC_5]$.

The scope-aided selection makes use of the same algorithm. The only difference is that the list of entities to be covered, which is provided as input for Algorithm 3, contains the set of in-scope entities identified by our approach, rather than the full set of entities available in the target program. When supported by the scope-aided approach, TC_3 is the first test case chosen by Algorithm 3 as it covers the highest number of in-scope statements (we recall that the second last line of Table 4.1 counts the number of in-scope statements covered by each test case). After TC_3 is selected, statement 7 is the only statement yet to be covered. TC_1 is the next test case chosen as it covers statement 7. After TC_1 is selected, 100% coverage is achieved for the

in-scope statements and the test case selection stops. The test suite produced by Algorithm 3 is then provided $TS_{s'} = [\mathrm{TC}_3, \mathrm{TC}_1]$.

4.2.3 Scope-aided Minimization

Test suite minimization is a technique that seeks to reduce as much as possible the test suite size by identifying and eliminating redundant test cases from it.

4.2.3.1 Definition of scope-aided minimization

More formally, test suite minimization [56] can be defined as follows:

Definition 15. (*Minimization Problem*):

***Given**: A program P; a test suite T; a set of entities $\mathcal{E} = \{e_1, ..., e_n\}$ that must be exercised to provide the desired test coverage of P; and subsets of T: $\{T_1, ..., T_n\}$, each one associated with one of the e_i such that any of the test cases $t_j \in T_i$ can be used to test e_i*

***Problem**: Find a representative set T' of test cases from T that satisfies all $e_i \in \mathcal{E}$*

The problem of scope-aided minimization is analogous to the minimization problem, but we aim at covering only the entities that belong to a given testing scope. The definition of Scope-aided Minimization, comparable to that of Minimization, is provided as follows:

Definition 16. (*Scope-aided Minimization Problem*):

***Given**: A program P; a test suite T; a set of entities $\mathcal{E} = \{e_1, ..., e_n\}$ that must be exercised to provide the coverage of P; a testing scope $\mathcal{S}$; a subset of in-scope entities $\mathcal{E}_s \in \mathcal{E}$; and subsets of T: $\{T_{s_1}, ..., T_{s_m}\}$, each one associated with one of the $e_i \in \mathcal{E}_s$ such that any of the test cases t_j belonging to T_{s_i} can be used to test e_i*

***Problem**: Find a representative set T' of test cases from T, such that $T' \in \mathcal{S}$, that satisfies all $e_i \in \mathcal{E}_s$*

The test suite minimization problem is equivalent to the well-known set covering problem which is NP-complete [71]. For this reason, the application of heuristics is encouraged. Chen and Lau [25] proposed two heuristics, namely GE and GRE, for the test suite minimization problem inspired on a previous heuristic from Harrold et al. [56]. In their experiments comparing these three heuristics, they observed that none of them was always the best. For our instantiation example and for our exploratory study we arbitrarily adopted the GE heuristic.

4.2.3.2 The scope-aided approach applied to a minimization example

As above stated, for this instantiation we adopted the GE heuristic from Chen and Lau [25]. The GE heuristic, depicted in Algorithm 4, is built upon a greedy algorithm based on the notions of *essential* test cases. A test case is said to be essential if it is the only test case that can cover a given entity e_i. Algorithm 4 first selects all essential test cases (first `while` loop starting at line 2); then, it repeatedly selects the test case that covers the maximum number of uncovered entities until all entities are covered (second `while` loop, line 7). In the original implementation, if there is a tie between several test cases, an arbitrary choice is made. Similar to previous examples, in this work our implementation of *getNextTestCase* (line 8) sorts the tied test cases according to their IDs to guarantee a deterministic output.

Algorithm 4: Minimization (adopting the GE heuristic proposed in [25])

```
   Input: T            /*the test suite from which test cases can be selected*/
          entities                              /*list of entities to be covered*/
          coverageInfo          /*list of entities covered by each test from T*/
   Output: T'          /*a subset of T, with which to test the target program*/
1  T' ⟵ [ ]                                  /*T' is initialized as an empty list*/
2  while thereAreEssentialTestCases(T, entities, coverageInfo) do
   |                                      /*essential test cases are selected first*/
3  |  add(essentialTestCase, T')
4  |  updateUncoveredEntities(entities, essentialTestCase)   /*removes the entities
   |   covered by the essential test case from the list of uncovered
   |   entities*/
   end

5  if isEmpty(entities) then
6  |  return T'
   else
7  |  while thereAreUncoveredEntities(T, entities, coverageInfo) do
8  |  |  selectedTestCase ⟵ getNextTestCase(T, entities, coverageInfo)
   |  |   /*selects the test case that covers the highest number of uncovered
   |  |   entities*/
9  |  |  add(selectedTestCase, T')
10 |  |  updateUncoveredEntities(entities, selectedTestCase)          /*removes the
   |  |   entities covered by the selected test case from the list of
   |  |   uncovered entities*/
   |  end
   end

11 return T'
```

When applied to the motivating example, Algorithm 4 receives the list of entities that need to be covered (1, 2, . . . , 11, 12) and identifies the essential test cases. The first essential test case is TC_2 as it is the only one that covers statement 3. TC_2 is selected and the list of uncovered entities is updated (2, 7, 10, 11, 12). The next essential test case selected is TC_1 as it is the only one that covers statement 7. After TC_1 is selected, the list of yet to be covered entities is (10, 11, and 12) because it covered two previously uncovered statements (2 and 7). This process continues with the further selection of TC_4 (as being an essential test case to cover statement 11) and then TC_5 (an essential test case to cover statement 12). After TC_5 is selected, all the entities have been covered and Algorithm 4 returns the minimized test suite $TS_m = [TC_2,TC_1,TC_4,TC_5]$. Notice that, for this small example, all the test cases happened to be selected as being *essential*, but this is not always the case.

When supported by the scope-aided approach, Algorithm 4 receives the list of the in-scope entities only. This is the only difference with respect to the traditional approach for minimizing test suites using the GE heuristic. The first test case selected by the scope-aided minimization is TC_1 because it is an essential test case to cover statement 7. After TC_1 is selected, there are no more essential test cases and the algorithm continues using the greedy additional approach. At this point, the entities that still need to be covered are (8 and 9) and TC_2 and TC_3 are tied as both cover these two entities. The tie is solved with TC_2 being selected and the minimization stops as all the in-scope entities have been covered. The scope-aided minimized test suite is then provided $TS_{m'} = [TC_1,TC_2]$.

4.3 Exploratory Study

We conducted an exploratory study to assess the usefulness of adopting the proposed coverage criterion for boosting test case prioritization, selection, and test suite minimization. In this section we discuss the settings of the study. More precisely, we focus on three research questions:

RQ1: (Scope-aided usefulness for test case prioritization) *Does scope-aided approach improve test case prioritization?* In particular, we will compare scope-aided prioritization with the original one (not scope-aided) with respect to fault detection rate when considering *in-scope faults* (RQ1.1) and *all faults* (RQ1.2);

RQ2: (Scope-aided usefulness for test case selection) *Does scope-aided approach improve test case selection?* In particular, we will compare scope-aided selection with the original one (not scope-aided) concerning the size of the selected test suite (RQ2.1) and the impact of the selection on the test suite's fault detection ability (RQ2.2);

RQ3: (Scope-aided usefulness for test suite minimization) *Does scope-aided approach improve test suite minimization?* In particular, we will compare scope-aided minimization with the original one (not scope-aided) with respect to the

effectiveness of the minimization (RQ3.1) and the impact of the minimization on the test suite's fault detection ability (RQ3.2).

4.3.1 Study Subjects

In order to carry out our exploratory study and to investigate our research questions in a realistic setting, we looked for subjects in the Software-artifact Infrastructure Repository (SIR) [28]. SIR contains a set of real, non-trivial programs that have been extensively used in previous research. For selecting our subjects, some prerequisites had to be considered: first, the subjects should be written in the C language; second, they should contain faults (either real faults or seeded ones) and a test suite associated with them.

Table 4.3: Details about the study subjects considered in our investigations

Sub.	Ver.	LoC	Test Suite	Mutant versions	Mutants compiled	Mutants killed by SIR suite	Hard to kill mutants
grep	v1	9463	199	7981	2350	325	288
grep	v2	9987	199	9471	2373	311	283
grep	v3	10124	199	9690	2425	305	272
grep	v4	10143	199	9821	2602	354	260
grep	v5	10072	199	9797	2603	374	271
gzip	v1	4594	195	5049	3965	580	354
gzip	v2	5083	195	6088	4742	499	333
gzip	v3	5095	195	4755	4261	547	249
gzip	v4	5233	195	4641	4119	593	362
gzip	v5	5745	195	5843	5072	502	335
sed	v1	5486	360	6548	1991	815	673
sed	v2	9867	360	3946	3055	919	793
sed	v3	7146	360	4125	1176	864	782
sed	v4	7086	363	7697	1838	900	815
sed	v5	13398	370	1967	1483	1000	914
sed	v6	13413	370	1889	1398	1000	925
sed	v7	14456	370	2151	1133	1000	919
	Total:	146391	4523	101459	46586	10888	8828

For this study we selected a total of 17 variant versions from three C subjects: grep, gzip, and sed. grep is a command-line utility that searches for lines matching a given regular expression in the provided file(s); gzip is a software application used for file compression and decompression; and sed is a stream editor that performs basic text transformations on an input stream. grep and gzip are available from SIR with 6 sequential versions (1 baseline version and 5 variant versions with seeded

faults) whereas sed contains 8 sequential versions (1 baseline version and 7 variant versions with seeded faults).

Other materials were used during this study: KLEE was used in one of the steps of our approach to determine the set of in-scope entities; gcov[3] and lcov[4] utilities were used for collecting accurate coverage metrics. MILU [68] was used to generate mutated versions of our study subjects. Finally, our own code was used to automate the majority of the steps followed during this study.

Table 6.1 displays additional details about our study subjects. Column "LoC" shows the lines of code[5] of each variant version. The 4th and 5th columns display the number of test cases available in the SIR test suite and the number of mutant versions created for each study subject, respectively. Please ignore for the moment the last three columns, whose meaning is explained later on.

For carrying out our study, we needed a way of identifying different testing scopes for the investigated subjects in order to evaluate the effectiveness of our scope-aided approach. The usage scenarios depicted in the next three sections represent the testing scopes for our study.

Testing scopes for grep

Because of its inherent characteristics, grep made this task of identifying the testing scope relatively easy as it already considers three major usage scenarios:

1. grep -G is the *default* behavior and it interprets the provided pattern as a basic regular expression.

2. grep -E switches grep into a special mode in which expressions are evaluated as extended regular expressions as opposed to its normal pattern matching. The essential difference between basic and extended regular expression is that some characters (e.g., '?', '+', '|', etc) have a special meaning when used in the extended mode whereas they are considered ordinary characters when used in the basic expression.

3. grep -F makes grep interpret the pattern provided as a list of fixed strings, separated by new lines, and performs an *OR* search without doing any special pattern matching.

Indeed, these scenarios are so commonly used that there are even shortcuts to them: *egrep* is equivalent to grep -E while *fgrep* corresponds to grep -F.

Testing scopes for gzip

For gzip, we defined the three following scenarios in which our subject could be used:

[3] `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
[4] `http://ltp.sourceforge.net/coverage/lcov.php`
[5] Collected using the CLOC utility (`http://cloc.sourceforge.net/`).

1. gzip is used, within a bigger system, for *compressing* files only;

2. gzip is used by an online service only for *decompressing* the files submitted by the service's users; and

3. gzip is used for *compressing* not only files but also whole directories *recursively*.

Testing scopes for sed

For sed, two usage scenarios were defined:

1. sed is used to perform basic text transformations in the contents provided in the standard input (i.e., sed scripts and input text are both provided in the standard input);

2. sed programs (or sed scripts) are used to parse the text from the input files provided.

4.3.2 Tasks and Procedures

In this study we considered three types of entities: function, statement, and branch, which correspondingly identify three coverage criteria. Then, for each version of the subjects investigated and for each type of entity we performed the following tasks:

1. Applied traditional prioritization, selection, and minimization techniques on the object's test suite

2. Applied our scope-aided prioritization, scope-aided selection, and scope-aided minimization on top of the traditional techniques

3. Evaluated, for each possible combination of testing scope and adequacy criteria, the performance of the scope-aided approach when compared to the original techniques and considering mutant faults.

As stated in step 3, in this study we considered mutant faults. For generating mutated versions of our study subjects we used MILU [68], a C mutation testing tool designed for both first order and higher order mutation testing that supports many mutant operators (e.g., statement deletion, constraints replacement, etc). For our study, we allowed MILU to generate first order mutants of our subjects using any of the available mutant operators.

In sections 4.4, 4.5, and 4.6, we report our results with respect to the set of *all faults* and *in-scope faults*. The identification of these sets is detailed next.

After the mutants are generated, we run the subject's test suite against the mutated versions to identify which mutants would be killed by the baseline test suite. These compose the set of *all faults* (mutants) considered for the next steps of

our study. The number of *all faults* is displayed in the column "Mutants killed by SIR suite" in Table 6.1.

The ideal way to decide whether a fault is relevant or not in a given scope would be to refer to failure reports from the field where the program is used under the scope constraints. We did not have such data for our subjects, so in the aim of having unbiased data, we created, for each variant version investigated and for each specific scope, one random test suite containing 1K test cases: for each scope, the set of *in-scope faults* is composed by those mutants that are killed by the random test suite targeting that specific scope.

For generating the random test cases we developed our own scripts following a strategy very similar to that of Balcer et al. [10] for test scripts generation based on the test specification language (TSL).

We start by identifying all the possible input parameters and environment flags that can influence the behavior of the study subject (e.g., for gzip, the `-d` and `--decompress` flags can be used for decompressing files; the `-S` and `--sufix` flags can be used to define the suffix to be used for compressed files; and so on). We then investigated which input data, if any, was required by the subject (e.g., gzip can receive, as input data, a file to be either compressed or decompressed) and created the necessary support files (for gzip we created directories containing multiple files to be compressed; multiple compressed files to be tested or decompressed; etc).

Our script creates random test cases by choosing arbitrary input variables and input data to be used by the subject. To make sure that the subject's behavior is explored to its maximum, we allow our script to generate test cases that exercise unexpected behavior and error inputs (such as trying to decompress a file that is not compressed; or trying to compress a non-existent file, for example). A random test case is accepted in the random test suite for a given scope if the input domain constraints of the scope hold (e.g., for scope 2 of gzip, a valid test case should always contain the `-d`, or `--decompress`, flag).

4.3.3 Metrics

In this section we introduce the metrics used in our exploratory study to assess the usefulness of our approach. The *APFD* metric is used to assess scope-aided prioritization whereas *test suite size* and *impact on fault detection capability* are used to evaluate scope-aided selection and minimization.

Average Percentage of Faults Detected (APFD)

In order to address the research questions concerning prioritization, we need a way to assess and compare scope-aided prioritization with other prioritization techniques. In this work, we adopt the APFD metric. APFD was first introduced in [115] and in its first definition it assumed all the test costs and faults severities to be uniform.

Later, a variant version of this metric incorporating varying test costs and different fault severities, the APFD_C, was introduced [32].

Because in our approach we consider that faults have different relevance within a given testing scope, our studies are naturally suited to the cost-cognizant version of the metric.

APFD_C is calculated according to Equation 4.2. In this equation, T is a test suite containing n test cases with costs t_1, t_2, ..., t_n; F is a set of m faults revealed by T with severities f_1, f_2, ..., f_m; and TF_i is the first test case of an ordering T' of T that reveals fault i.

$$\text{APFD}c = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TFi}^{n} t_j - \frac{1}{2}t_{TFi}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{4.2}$$

In this work, we consider different values of severity according to fault relevance, since we focus in assessing the contribution of scope-aided prioritization to increase the speed at which in-scope faults are revealed. On the other hand, for simplicity, we consider all the tests to have the same cost, since we are not focusing on the cost of test cases.

Test Suite Reduction

The test suite reduction ratio achieved by test case selection or test suite minimization is measured according to Equation 4.3.

$$\text{Reduction} = \left(1 - \frac{\#\ \textit{test cases in the reduced test suite}}{\#\ \textit{test cases in the original test suite}}\right) \times 100\% \tag{4.3}$$

We apply this formula both to scope-aided selection and minimization approaches, and to the original (not scope-aided) ones.

Impact on Fault Detection Capability

The impact on fault detection capability of a given test suite is calculated according to Equation 4.4.

$$\text{Impact} = \left(1 - \frac{\#\ \textit{faults detected by the reduced test suite}}{\#\ \textit{faults detected by the original test suite}}\right) \times 100\% \tag{4.4}$$

As above, we apply this formula both to scope-aided selection and minimization approaches, and to the original (not scope-aided) ones.

4.3.4 Execution

Many of the mutants generated by Milu [68] would not compile and this represented a first filtering to reduce the amount of computation required to define the sets of

all faults and *in-scope faults* (the number of mutants that compiled for each study subject is displayed in the 6th column of Table 6.1). However, for some subjects, the number of mutants that compiled was still too much. For gzip, for example, the average number of compiled mutants for each version was 4431. We then decided that, for each variant version investigated, we would run the baseline test suite (the one from SIR) on a set of 1K randomly selected mutants.

When evaluating the test suites, we noticed that one of the test cases available for gzip requires the compression of a 2GB file and, even after our decision of selecting "only" 1K mutants per version, we would still need to run that test case 5k times. For this reason, we decide to remove this test cases and a few others that would add a lot of computation overhead in our already expensive study, without contributing with significant added value. The final test suite for gzip considered in our study consisted of 195 test cases.

For running the random test suites to identify the set of in-scope faults, we are interested only in those mutants that could be killed by the baseline suite (column "Mutants killed by SIR suite" of Table 6.1). However, after a preliminary analysis of this set, we noticed that for some subjects, in particular for gzip, many mutants could be killed by the vast majority of the test cases, and a considerably amount of them, even by any test case. Because in our study we want to evaluate the ability of the scope-aided approach to identify the most relevant faults to a given testing scope, we followed the suggestions given in [7] and decided to eliminate the easy-to-kill mutants before proceeding to the next steps. We considered only the mutants that could not be detected by more than 50% of the test cases (these are displayed in the last column of Table 6.1). After this filtering, we proceeded with the execution of the random test suites.

As the reader may have noticed already, these steps required a lot of computational effort. About 4.5 million tests were run over the mutated versions only for the identification of the set of *all faults* (199 tests from grep $\times$ 5000 mutants + 195 tests from gzip $\times$ 5000 mutants + $\sim$ 365 tests from sed $\times$ 7000 mutants). After this step, we still needed to run the random test suites (3k tests for grep and gzip, and 2k tests for sed) over the set of hard-to-kill mutants. Obviously, however, these steps are not required for the adoption of our approach as they were applied only for supporting our empirical studies.

4.4 Prioritization Study (RQ1)

As previously stated, prioritization aims at reordering a test suite so that potential faults are revealed faster. A natural question that arises, thus, is whether scope-aided prioritization helps traditional approaches to provide faster detection of faults in reuse testing. In this section we investigate the usefulness of scope-aided approach to support the test case prioritization task with respect to the following research questions:

RQ1.1: *In-scope faults detection rate*: how does scope-aided prioritization compare with original (not scope-aided) prioritization with respect to fault detection rate when considering *in-scope faults*?

RQ1.2: *All faults detection rate*: how does scope-aided prioritization compare with original (not scope-aided) prioritization with respect to fault detection rate when considering *all faults*?

We answer these questions by comparing, for each prioritization approach considered in this study, the original approach with the scope-aided one in terms of the weighted average of the percentage of faults detected, or $APFD_C$, over the life of a given test suite.

With the purpose of having a broader view of the extent to which the scope-aided approach could impact the traditional prioritization techniques, we wanted to apply it to different approaches to prioritization: we focused on coverage-, similarity-, and search-based strategies, which represent most of the used approaches. To select the specific techniques among those found in the literature, we used three criteria: the technique should be well documented so to be properly reproduced; it should be proven to be effective; and, given the size of the studies, it should be computationally affordable. In the end, we selected two coverage-based prioritization approaches (the well-known *total* and *additional* greedy heuristics); one similarity-based approach (proposed by Jiang et al. [69]); and one search-based technique (one instantiation of the Hill Climbing algorithm applied to test prioritization as proposed in the work from Li et al. [79]).

4.4.1 RQ1.1: Rate of Faults Detected (*in-scope faults*)

To answer RQ1.1, we assess scope-aided prioritization in comparison with the traditional techniques when considering the in-scope faults. For calculating the $APFD_C$ values, we assign 0 as the severity of the out-of-scope faults to consider only the in-scope ones, getting severity equal to 1 ($APFD_C(1,0)$).

All the results are reported in the Figures 4.1, 4.2 and 4.3, for the three coverage criteria considered.

Considering the overall average for each subject, the scope-aided approach improved the $APFD_C$ for all the subjects when considering the coverage-based approaches, with the biggest improvement being an increase of 40.35% in the average $APFD_C$ for gzip using the total approach (from 49.05 to 68.84). The smallest improvement happened for gzip using the additional approach, but in that case the scope-aided approach achieved only a negligible improvement of less than 1% in the overall average $APFD_C$. Concerning the similarity-based approach, scope-aided improved the $APFD_C$ for grep and gzip, and it was almost tied with the traditional technique for sed (overall average $APFD_C$ of 93.46 for the traditional technique; and 93.29 for the scope-aided one). The biggest increase in the overall average $APFD_C$

was a 10.12% improvement for gzip. With respect to the search-based approach, scope-aided was defeated by the traditional technique when considering gzip; it was basically tied when considering sed (overall average $APFD_C$ of 95.83 for the traditional technique; and 95.82 for the scope-aided one); and it slightly improved (1.33%) the overall average $APFD_C$ for grep.

Figure 4.1 displays the impact of the adoption of scope-aided approach with respect to the *Function* coverage criterion. Each subfigure displays the consolidated results related to a given approach (either total, additional, similarity, or search-based) applied to one of the subjects investigated in our study (grep, gzip, or sed). The x-axes represent the versions of each subject (recall that we analyzed all the versions of our study subjects available from SIR, i.e., 5 versions for grep and gzip, and 7 versions for sed), while the y-axes display the impact on the $APFD_C$ metric. The height of the vertical bars represents the contribution achieved by the adoption of the scope-aided approach for each testing scope. In particular, for similarity and search-based the contribution is obtained as an average of 50 runs, to account for randomness in these techniques.

For inter-graph comparison, observe that (besides the bar height) the values of the y-axes should also be considered as the scale of that axis changes across the graphs. Let us consider, for example, Figures 4.1b and 4.1d: even though the biggest bars in each graph have more or less the same size, they represent very different values of the contribution to the $APFD_C$ (approximately 90 in Figure 4.1b, and approximately 2 in Figure 4.1d).

For grep and gzip the leftmost bar (dark grey) is related to scope 1; the central bar (grey) is associated with scope 2; and the rightmost bar (light grey) represents the scope 3. For sed, because we investigated two testing scopes, we have only two bars for each version: scope 1, represented by the leftmost bar (dark grey); and scope 2, associated with the rightmost bar (light grey). Bars with negative values mean that the adoption of scope-aided approached achieved an $APFD_C$ lower than the one achieved by the traditional prioritization approach (not scope-aided). If a bar is not visible it is because, for that case, our approach was tied with the traditional technique.

Figures 4.2 and 4.3 are analogous to Figure 4.1 and they display the consolidated data with respect to the *Statement* and *Branch* coverage criteria, respectively.

As regards the function coverage criterion (Figure 4.1), the scope-aided approach improved the original $APFD_C$ in 91 cases; it was tied with the traditional technique in 12 cases; and it was defeated 73 times. The highest improvement was achieved for gzip v1 when adopting the total prioritization approach for the scope 2 (Figure 4.1b). The scope-aided approach obtained an $APFD_C$ of 98.57 against 8.78 achieved by the traditional technique. The greatest loss, on the other hand, occurred for gzip v4 when applying the search-based prioritization for the scope 1 (Figure 4.1k); the traditional approach achieved an $APFD_C$ of 80.13 and the scope-aided one achieved 73.86. Overall, the average $APFD_C$ achieved by scope-aided prioritization (88.84) was higher than the one achieved by the original techniques (86.36).

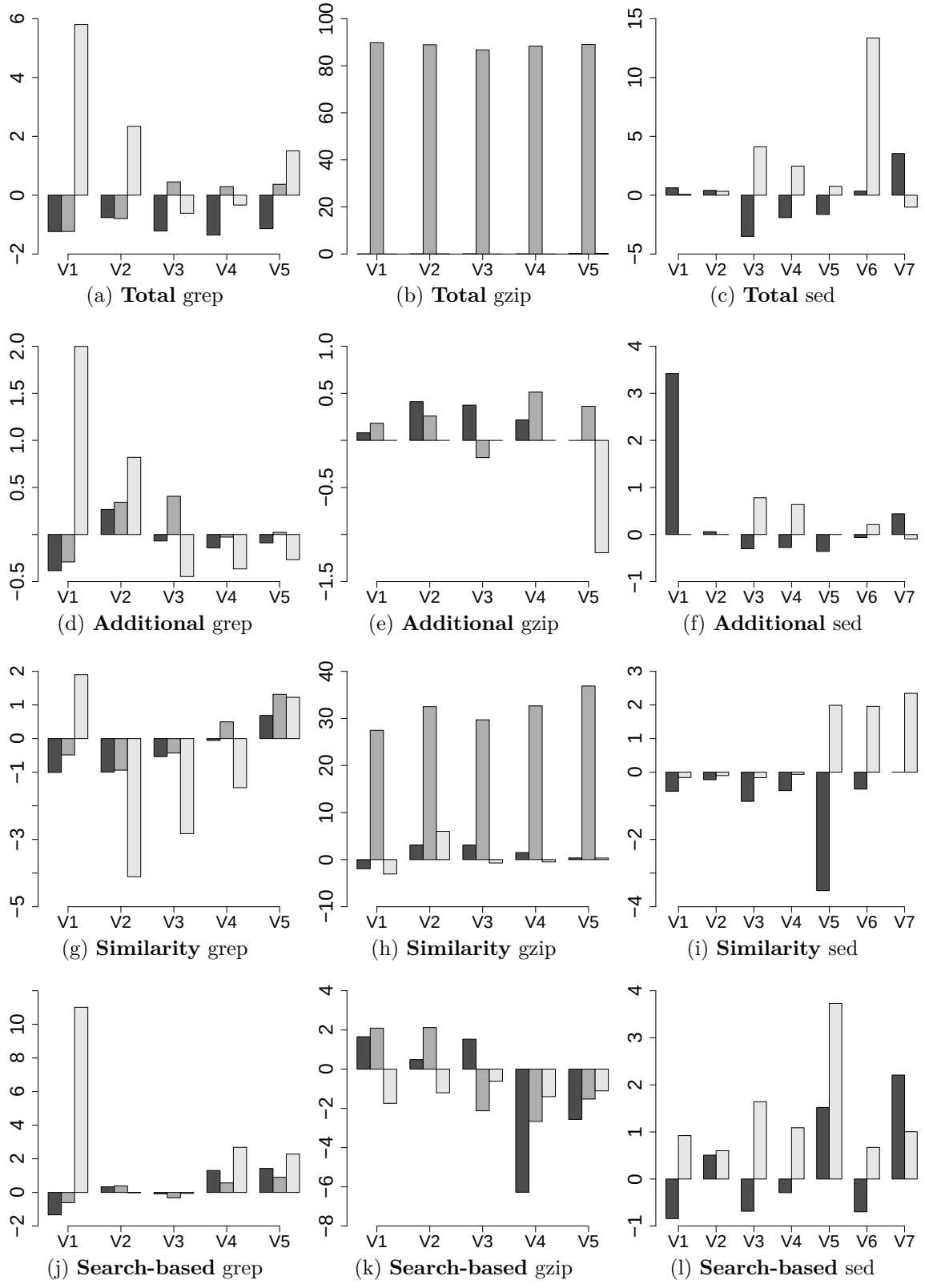


Figure 4.1: Contribution to the $APFD_C$ when considering the set of ***in-scope faults*** and the ***Function*** coverage criterion

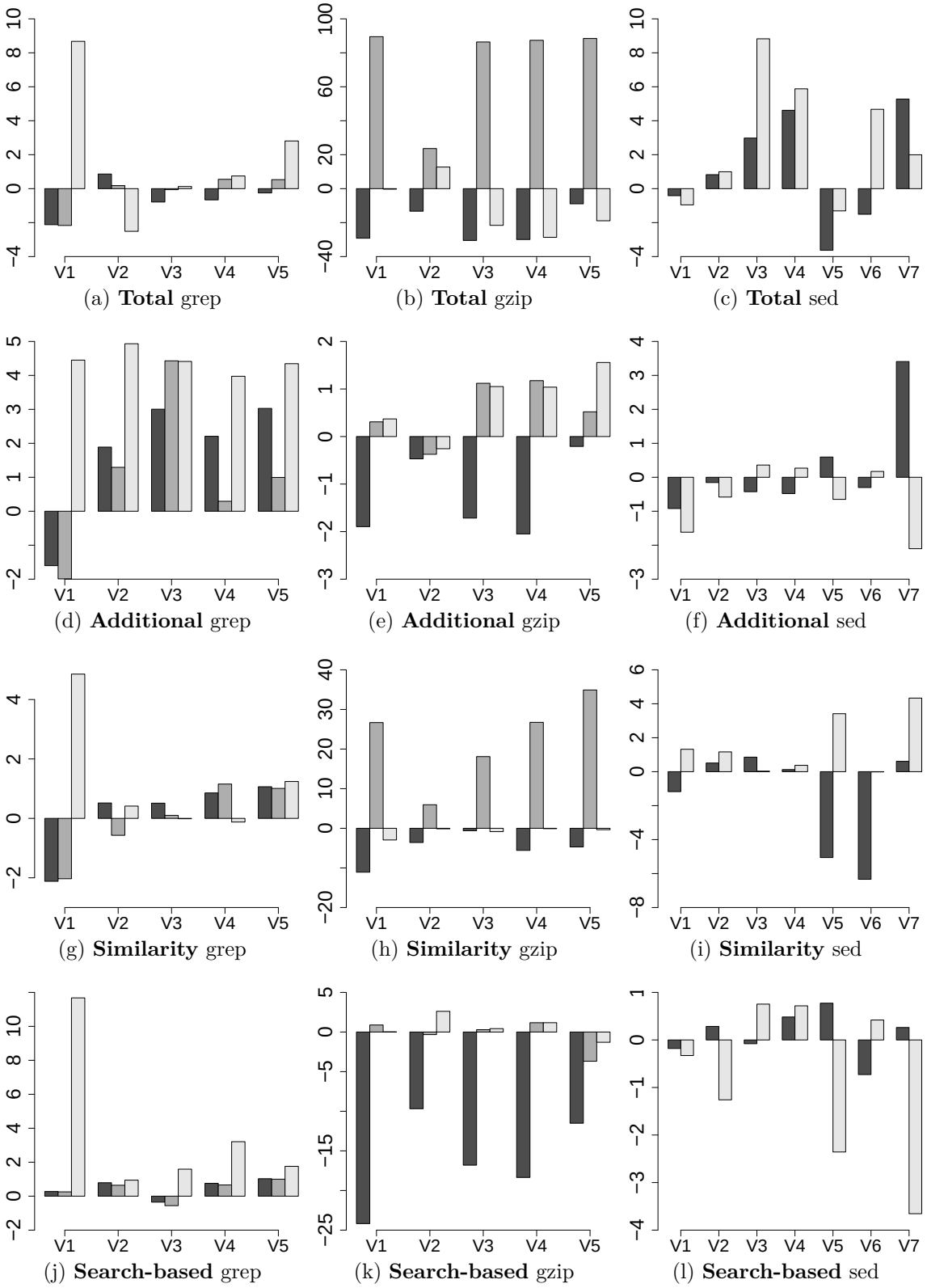


Figure 4.2: Contribution to the $APFD_C$ when considering the set of ***in-scope faults*** and the ***Statement*** coverage criterion

For the statement coverage criterion (Figure 4.2), once again the scope-aided approach performed better than the traditional one in the majority of the cases (101 times against 75); it also achieved a better average $APFD_C$ (88.55) than the traditional approaches (86.82). This time, both the best and the worst results were associated with the same combination of subject and prioritization strategy. The greatest loss was noticed for gzip v3 with the total algorithm for scope 1 (46.31 against 76.80), whereas the highest improvement was seen for the version 1 and scope 2 of gzip (Figure 4.2b). The scope-aided test suite yielded an $APFD_C$ of 98.71 against 9.21 achieved by the traditional technique.

The scope-aided approaches also outperformed the traditional ones when looking at the overall results from the perspective of the branch coverage criterion (Figure 4.3). The results were similar to the ones obtained for the statement coverage criterion and, again, the best and worst results were associated with the total algorithm being applied for prioritizing the test suites from gzip. The highest improvement in the $APFD_C$ was observed for the version 1 and testing scope 2 (improved from 9.86 to 98.63); whereas the worst result was seen for the version 1 and scope 2 (26.09 against 75.13). All over, the scope-aided approach improved the original $APFD_C$ 94 times; it was tied with the traditional technique in 1 single case; and it was defeated 81 times. Besides that, it achieved a higher average $APFD_C$ value (88.71 against 86.72 achieved by the traditional techniques).

One consistent result across all the coverage criteria was the fact that the biggest discrepancy between results was always related to the total approach being applied for the gzip subject. One possible explanation for that is the fact that the test suite for gzip contains many test cases that are very similar to each other. Many test cases related to the compression task, for example, are repeated different times for each different compression level supported (from 1 to 9), which in the end may have a very small impact in the set of entities exercised by those test cases (or may even not impact it at all). Recall that the total strategy will prioritize the test cases according to the total number of entities covered by them and test cases that are very similar to each other will probably cover the same amount of entities. Besides that, there is a big chance that very similar test cases will reveal the same set of faults. When the total approach puts these "groups" of test cases all together, two main things could happen that may have a big impact in the $APFD_C$: either all test cases in a given group reveal many faults (in such case, the first test case in the group will contribute immediately to the increase of the $APFD_C$) or none of the cases in the group reveals any fault (this would *freeze* the contribution to the $APFD_C$ metric until all test case in that group are evaluated).

We manually evaluated the set of in-scope faults assigned for each scope and for all the versions of gzip and confirmed that specially for the scope 2, we had a situation in which a few test cases would reveal a big set of faults whereas the vast majority of the test cases would either reveal just a small set of faults or not reveal any fault at all. In such scenario, a good (or bad) choice of the first test case can have a huge impact in the $APFD_C$ achieved by the prioritized test suite.

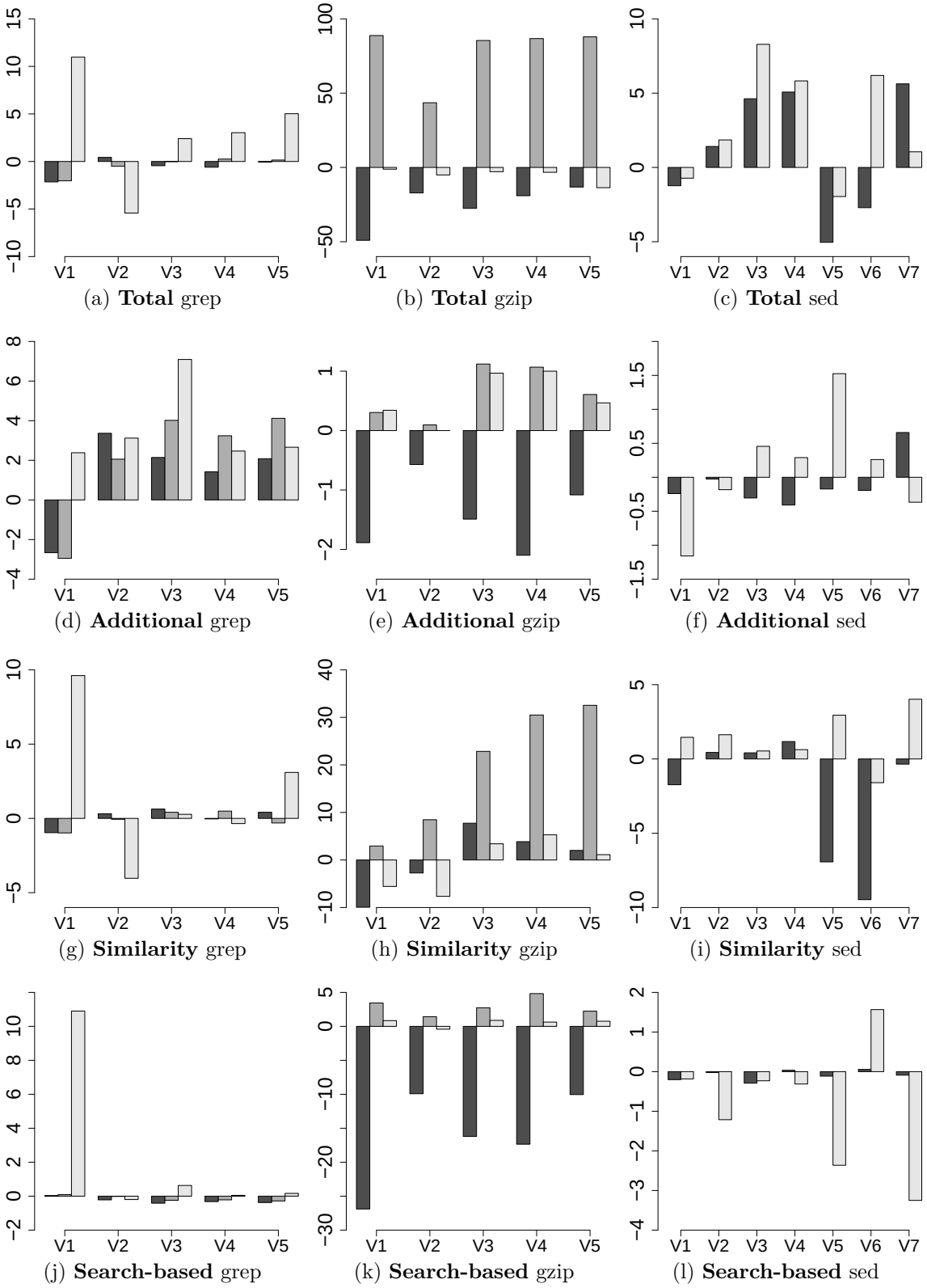


Figure 4.3: Contribution to the $APFD_C$ when considering the set of ***in-scope faults*** and the ***Branch*** coverage criterion

Because test case prioritization seeks to order the test cases in a way to maximize the benefits even if the testing needs to be prematurely halted at some point, we also investigated the results achieved by the scope-aided prioritization when considering different fractions of a given test suite (75%, 50%, and 25%).

Table 4.4: Average $APFD_C(1,0)$ (and coefficient of variation) when considering different fractions of the prioritized suites

Coverage criterion	**Fraction: 75%**		**Fraction: 50%**		**Fraction: 25%**	
	original	scope-aided	original	scope-aided	original	scope-aided
Function	88.0 (0.13)	**88.6** (0.11)	87.6 (0.13)	**88.2** (0.10)	85.0 (0.13)	**85.1** (0.10)
Statement	88.9 (0.11)	88.8 (0.12)	86.2 (0.13)	**87.9** (0.10)	84.4 (0.13)	**86.2** (0.09)
Branch	89.0 (0.10)	88.3 (0.13)	87.9 (0.10)	87.3 (0.12)	85.1 (0.10)	**86.4** (0.09)
Average:	88.6	88.6	87.2	**87.8**	84.8	**85.9**

The results obtained are displayed in Table 4.4. The values provided are grouped by coverage criterion and they represent the average $APFD_C$ achieved for the different fractions of the prioritized test suites. The values enclosed in parentheses represent the coefficient of variation[6]. We highlight in bold the cases in which scope-aided prioritization performed better than the original (not scope-aided) prioritization.

Overall, scope-aided prioritization performed better than the original one in the majority of the cases, except when considering the fractions 50% and 75% for branch and the fraction 75% for statement where not scope-aided performed better.

Even if our approach outperformed the traditional ones in the majority of the cases, in Table 4.4 we can only see small improvements in the average $APFD_C$. It is important to observe though that for our experiments we adopted state-of-the-art prioritization approaches that already achieved very high $APFD_C$ values when applied in the traditional way. Thus, the scope-aided boost could improve on top of approaches that were already good.

With the purpose of better understanding the contribution provided by the scope-aided approach for the prioritization task, we also evaluated the results of our study from the perspective of the different prioritization strategies investigated. The results are displayed in Table 4.5 and the values provided are grouped by the combination of prioritization approach and coverage criterion.

When considering the approaches total, additional, and similarity, scope-aided prioritization performed better than the original ones for all the coverage criteria

[6]The coefficient of variation (CV), also known as relative standard deviation (RSD), is the ratio of the standard deviation σ to the mean μ. Different from the standard deviation that must always be understood in the context of the mean, the coefficient of variation allows the comparison between data sets with different means and even different units. The lower the value of CV, the lower the variance of that data set. Looking at Table 4.4, for example, we can tell that the combinations statement–25% and branch–25%, both with a CV of 0.09, had the lower variance across all the possible combinations of coverage criteria and test suite fractions.

Table 4.5: Average $APFD_C(1,0)$ (and coefficient of variation) when considering different prioritization approaches and different coverage criteria

Approach	Function		Statement		Branch	
	original	scope-aided	original	scope-aided	original	scope-aided
Total	77.0 (0.35)	**87.6** (0.14)	75.6 (0.34)	**81.0** (0.24)	74.2 (0.34)	**80.6** (0.24)
Additional	92.1 (0.07)	**92.3** (0.07)	94.1 (0.06)	**94.9** (0.05)	94.7 (0.05)	**95.5** (0.04)
Similarity	83.6 (0.18)	**87.3** (0.10)	86.1 (0.13)	**88.1** (0.08)	86.4 (0.12)	**88.5** (0.06)
Search-based	89.8 (0.08)	**90.2** (0.08)	91.6 (0.06)	90.2 (0.08)	91.6 (0.05)	90.2 (0.08)
Average:	85.7	**89.4**	86.8	**88.5**	86.7	**88.7**

considered. Concerning the search-based approach, the results from scope-aided prioritization were better for function, and worse for statement and branch.

4.4.2 RQ1.2: Rate of Faults Detected (*all faults*)

To answer RQ1.2, we assess scope-aided prioritization in comparison with traditional techniques when considering all faults. We calculated $APFD_C$ assigning the same severity for all the faults. Because we also consider all the test cases to have the same costs, the $APFD_C$ formula reduces to the traditional APFD in which costs are not considered.

Overall, the results achieved by the scope-aided approach when considering *all faults* were slightly worse than the ones achieved when considering only the in-scope faults, but still very competitive with the traditional techniques. This was a positive result to us: we were somewhat prepared to see a greater lose in the effectiveness when considering all the faults given that our approach targets the in-scope ones.

For the sake of space, we do not replicate the barplots that would look similar to the ones provided for RQ1.1. Instead, we highlight below the main results.

When considering the average among all the subjects, the average $APFD_C$ achieved by scope-aided (87.25) was better than the one achieved by the traditional approaches (86.54), but to a smaller extent when compared with the results achieved in the study considering in-scope faults. The biggest improvement observed was an increase of 16.5% in the average $APFD_C$ for gzip using the total approach (from 57.76 to 67.29), whereas the biggest defeat was a reduction of 4.74% in the average $APFD_C$ for gzip using the search-based approach.

Table 4.6 displays the average $APFD_C$ achieved by the scope-aided and the traditional approaches grouped by prioritization strategy and coverage criterion. Scope-aided prioritization performed always better than the corresponding original approach when considering the total strategy; and it performed always worse when considering the additional strategy; for similarity and search-based strategies the results varied. Even though the traditional approaches defeated the scope-aided

ones in a bigger number of cases, scope-aided still achieved better results when considering the overall average.

Table 4.6: Average $APFD_C(1,1)$ (and coefficient of variation) when considering different prioritization approaches and different coverage criteria

Approach	Function		Statement		Branch	
	original	scope-aided	original	scope-aided	original	scope-aided
Total	74.4 (0.24)	**78.4** (0.23)	73.8 (0.23)	**76.0** (0.26)	70.6 (0.31)	**74.7** (0.30)
Additional	92.9 (0.07)	92.6 (0.07)	95.5 (0.05)	95.4 (0.04)	96.2 (0.04)	95.9 (0.03)
Similarity	84.3 (0.11)	**86.5** (0.09)	85.9 (0.08)	85.7 (0.10)	87.0 (0.06)	86.5 (0.10)
Search-based	91.5 (0.04)	**91.6** (0.04)	93.1 (0.04)	91.6 (0.04)	92.8 (0.03)	91.5 (0.03)
Average:	85.8	**87.3**	87.1	**87.2**	86.6	**87.1**

4.5 Selection Study (RQ2)

In this section we investigate the usefulness of scope-aided approach to support the test case selection task with respect to the following research questions:

RQ2.1: *Test suite reduction*: how does scope-aided selection compare with the original one (not scope-aided) in terms of test suite reduction achieved?

RQ2.2: *Impact on fault detection capability*: what is the impact of scope-aided selection with respect to the test suite's fault detection capability when compared to the original (not scope-aided) selection and considering both *all faults* and *in-scope faults*?

4.5.1 RQ2.1: Test Suite Reduction

To answer RQ2.1, we assess scope-aided selection in comparison with the traditional technique in terms of the test suite reduction rate achieved by each approach. The reduction rate is calculated according to Equation 4.3 (detailed in Section 4.3.3).

Table 4.7 displays the average reduction achieved and the results are grouped by the different versions of each subject evaluated in our study. Scope-aided outperformed the traditional approach in all the cases considered with an average extra reduction of 10.69% for grep, 5.60% for gzip, and 3.94% for sed.

The smallest reduction achieved by the traditional selection was 65.33% and the biggest one was 98.61%; for scope-aided selection, the smallest reduction was 74.37% and the biggest one was 99.72%. In both cases, the smallest reduction was associated with grep while targeting the branch coverage criterion, and the biggest reduction was related to sed while targeting the function coverage criterion.

Table 4.7: Comparison between the average test suite reduction (and coefficient of variation) achieved by the scope-aided selection and the traditional approach

Subject versions	*grep*		*gzip*		*sed*	
	original	scope-aided	original	scope-aided	original	scope-aided
V1	77.7% (0.17)	**87.2%** (0.11)	91.6% (0.02)	**97.3%** (0.01)	93.6% (0.04)	**97.3%** (0.02)
V2	77.6% (0.17)	**88.7%** (0.08)	91.6% (0.02)	**97.0%** (0.02)	93.9% (0.04)	**98.1%** (0.02)
V3	77.9% (0.17)	**88.7%** (0.10)	91.8% (0.02)	**97.5%** (0.01)	93.5% (0.04)	**97.2%** (0.02)
V4	78.1% (0.16)	**89.0%** (0.09)	91.6% (0.02)	**97.3%** (0.02)	93.2% (0.04)	**97.2%** (0.02)
V5	78.1% (0.16)	**89.2%** (0.09)	91.8% (0.03)	**97.3%** (0.02)	93.3% (0.04)	**97.3%** (0.02)
V6	-	-	-	-	93.5% (0.04)	**96.6%** (0.03)
V7	-	-	-	-	92.9% (0.04)	**97.7%** (0.01)
Average:	77.9%	**88.6%**	91.7%	**97.3%**	93.4%	**97.4%**

When looking from the perspective of the different coverage criteria considered, the smallest rates of reduction achieved were 93.85% for the traditional approach and 95.98% for the scope-aided one (for function); 72.86% for the traditional approach and 78.89% for the scope-aided one (for statement); and 65.33% for the traditional approach and 74.37% for the scope-aided one (for branch).

The scope-aided approach was expected to perform better in this metric because, by construction, the test case selection is targeted to a subset of the (testing) input domain. In fact, RQ2.1 was a *proof of concept* that demonstrated preliminary positive results regarding the adoption of scope-aided approach for test case selection. We now proceed to investigate what is the impact on the fault detection capability caused by the test suite reduction.

4.5.2 RQ2.2: Impact on Fault Detection Capability

We answer RQ2.2 by assessing the scope-aided selection in comparison with the traditional technique in terms of the impact on fault detection capability generated by each approach. Such impact is calculated according to Equation 4.4 (detailed in Section 4.3.3). For this research question we considered both the set of *all faults* and the *in-scope faults*.

Figure 4.4 displays the boxplots for each approach grouped by the different coverage criteria considered. The y-axis displays the impact (in %) on fault detection capability. For this metric, the lower the impact, the better.

Overall, the traditional approach defeated the scope-aided one in all the cases. On average, the scope-aided generated an extra impact of 6.77%, 12.16%, and 13.84% for function, statement, and branch, respectively. Due to the fact that our data could not be assumed to be normally distributed, we adopted a non-parametric statistical hypothesis test, the Wilcoxon signed-rank test, to assess the null hypothesis that the difference between the impact on fault detection capability for the two

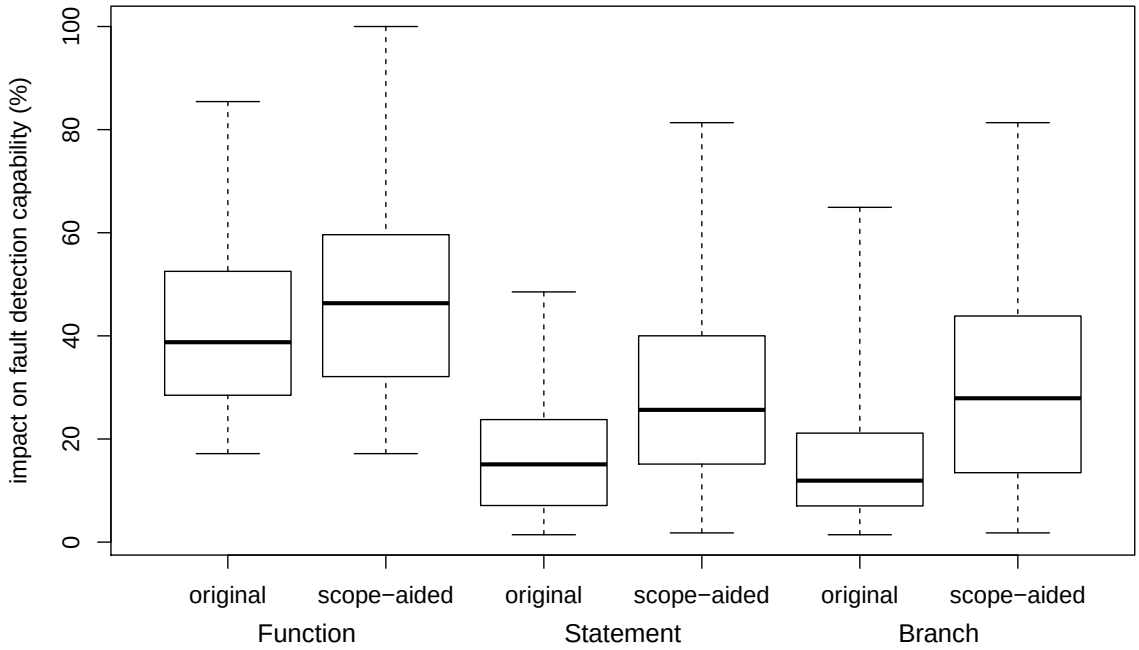


Figure 4.4: Impact on fault detection capability for the different coverage criteria when considering the set of *all faults*

approaches follows a symmetric distribution around zero, i.e., the null hypothesis is that the median values are statistically equivalent.

The resulting *p-values* for the Wilcoxon test were 0.02006, 0.0003549, and 1.687e−06, for function, statement, and branch, respectively. The null hypothesis was thus rejected in all the cases, which means that the observed differences are statistically significant at least at the 95% confidence level.

These results were in accordance with our initial intuition as we did not expect the scope-aided approach to generate a lower impact on the fault detection capability than the traditional one when considering all the faults.

When considering the set of in-scope faults (Figure 4.5), we can observe that the scope-aided approach improved with respect to the evaluation considering all the faults. Indeed, the median values achieved for the statement and branch coverage criteria were smaller than those achieved by the traditional approach. Once again, we applied the Wilcoxon signed-rank test and, this time, the null hypothesis was not rejected in any of the cases, i.e., the median values are statistically equivalent. The resulting *p-values* for the Wilcoxon test were 0.7421, 0.8139, and 0.9388, for function, statement, and branch, respectively.

Being statistically tied with the traditional approach is, in fact, a good result considering the fact that the scope-aided approach, targeting only a subset of the entities (and, consequently, producing much smaller test suites as we can see in Table 4.7), had the same impact on fault detection capability as the one suffered by the traditional technique targeting all the entities.

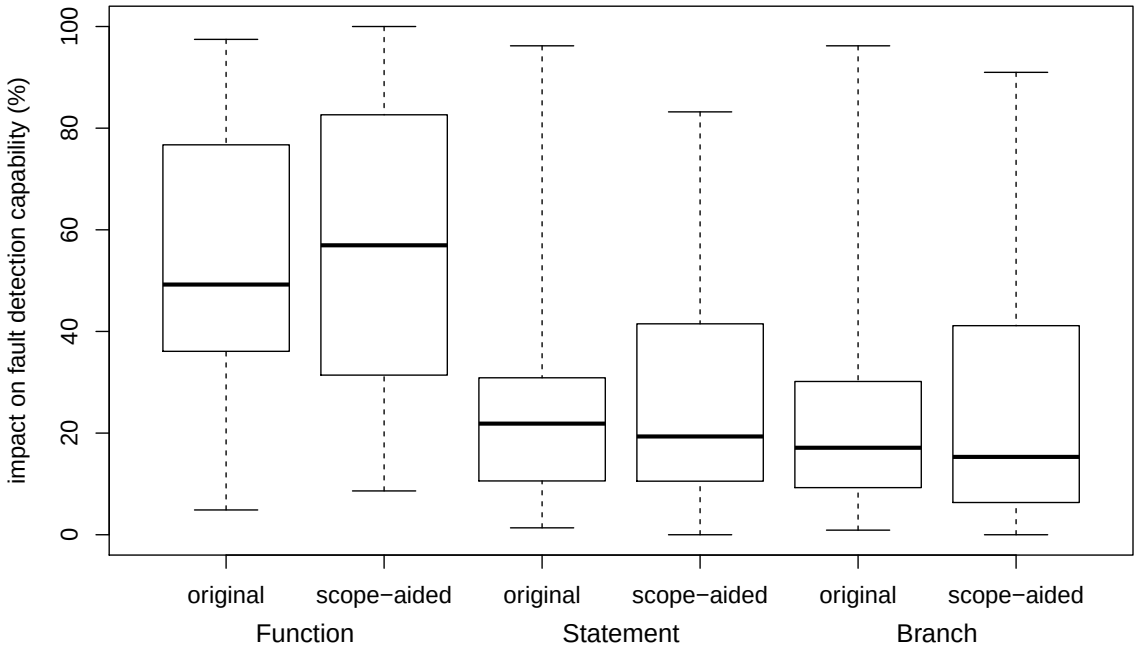


Figure 4.5: Impact on fault detection capability for the different coverage criteria when considering the set of *in-scope faults*

Looking from the perspective of the different subjects investigated, scope-aided approach had an average impact of 32.85% (against 37.30% generated by the traditional) and 35.68% (against 43.05% generated by the traditional) for grep and gzip, respectively. For sed, on the other hand, scope-aided generated an average impact of 39.70%, whereas the traditional approach generated an impact of 26.76%.

4.6 Minimization Study (RQ3)

In this section we finally investigate the usefulness of scope-aided approach to support the test case minimization task. The research questions that we address are the same as the set of RQ2.1 and RQ2.2 already considered for selection, except that here we refer to the minimization task:

RQ3.1: *Test suite reduction*: how does scope-aided minimization compare with the original one (not scope-aided) in terms of test suite reduction achieved?

RQ3.2: *Impact on fault detection capability*: what is the impact of scope-aided minimization with respect to the test suite's fault detection capability when compared to the original (not scope-aided) minimization and considering both *all faults* and *in-scope faults*?

4.6.1 RQ3.1: Test Suite Reduction

We answer RQ3.1 in an analogous way to that applied for RQ2.1 (Section 4.5.1) and we assess scope-aided minimization in comparison with the traditional technique in terms of the test suite reduction rate achieved by each approach.

Table 4.8 displays the average reduction achieved and the results are grouped by the different versions of each subject evaluated in our study. Scope-aided outperformed the traditional approach in all the cases considered with an average extra reduction of 10.53% for grep, 5.57% for gzip, and 3.69% for sed.

Table 4.8: Comparison between the average test suite reduction (and coefficient of variation) achieved by the scope-aided minimization and the traditional approach

Subject versions	***grep***		***gzip***		***sed***	
	original	scope-aided	original	scope-aided	original	scope-aided
V1	77.7% (0.17)	**87.4%** (0.11)	91.6% (0.02)	**97.3%** (0.01)	94.1% (0.04)	**97.5%** (0.02)
V2	77.7% (0.17)	**88.9%** (0.09)	91.8% (0.02)	**97.1%** (0.02)	93.9% (0.04)	**98.1%** (0.02)
V3	78.4% (0.16)	**88.9%** (0.10)	91.8% (0.02)	**97.5%** (0.01)	93.8% (0.03)	**97.3%** (0.02)
V4	78.6% (0.16)	**89.2%** (0.09)	91.8% (0.02)	**97.4%** (0.01)	93.4% (0.04)	**97.3%** (0.02)
V5	78.6% (0.16)	**89.3%** (0.09)	91.8% (0.03)	**97.3%** (0.02)	93.8% (0.04)	**97.2%** (0.02)
V6	-	-	-	-	93.8% (0.04)	**96.6%** (0.03)
V7	-	-	-	-	93.3% (0.04)	**97.8%** (0.01)
Average:	78.2%	**88.7%**	91.8%	**97.3%**	93.7%	**97.4%**

The smallest reduction achieved by the traditional minimization was 65.83% and the biggest one was 98.61%; for scope-aided minimization, the smallest reduction was 74.37% and the biggest one was 99.72%. In both cases, the smallest reduction was associated with grep while targeting the branch coverage criterion, and the biggest reduction was related to sed while targeting the function coverage criterion.

When looking from the perspective of the different coverage criteria considered, the smallest rates of reduction achieved were 94.36% for the traditional approach and 96.48% for the scope-aided one (for function); 72.86% for the traditional approach and 78.89% for the scope-aided one (for statement); and 65.83% for the traditional approach and 74.37% for the scope-aided one (for branch).

As previously explained in Section 4.5.1, the scope-aided approach was expected to perform better in this metric. We now evaluate the impact of the test suite reduction on the fault detection capability.

4.6.2 RQ3.2: Impact on Fault Detection Capability

To answer RQ3.2, we assess the scope-aided minimization in comparison with the traditional technique in terms of the impact on fault detection capability generated by each approach. This question is analogous to RQ2.2 (Section 4.5.2) and we

use the same equation (Equation 4.4) to calculate the impact on fault detection capability of a given test suite.

Figure 4.6 displays the boxplots for each approach grouped by the different coverage criteria considered. The results slightly improved with respect the ones obtained for the same metric in our study with test case selection (Figure 4.4, Section 4.5.2).

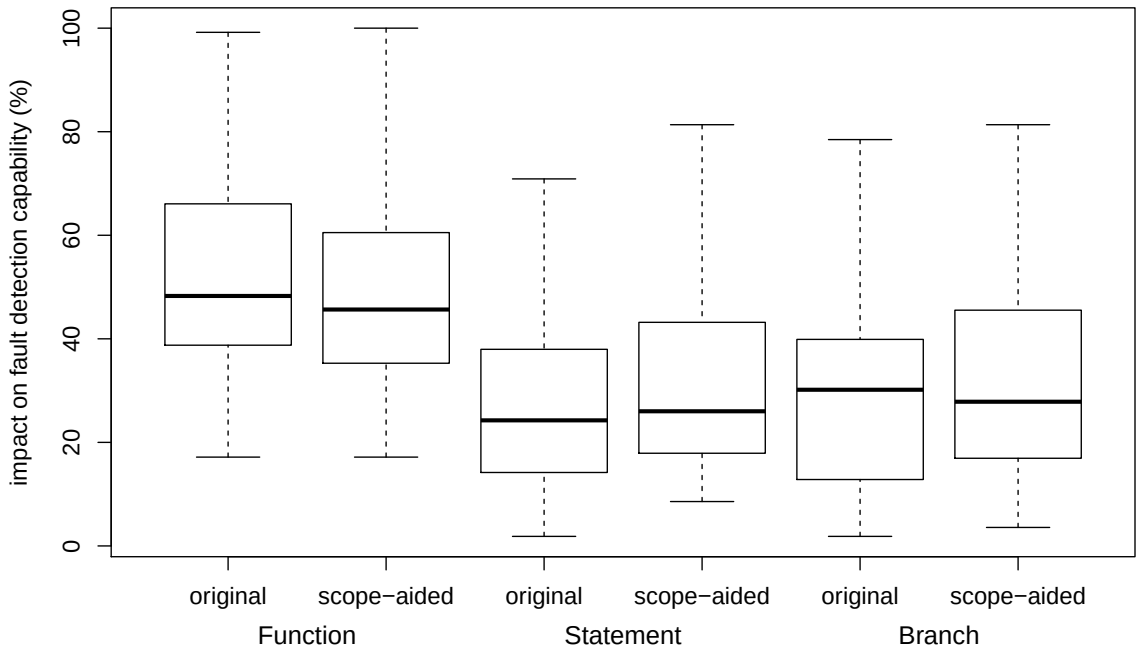


Figure 4.6: Impact on fault detection capability for the different coverage criteria when considering the set of *all faults*

By looking at the boxplots we can see that the scope-aided approach had a lower median value than the traditional technique for function and branch, and a higher one for the statement criterion. For evaluating whether the observed differences were statistically significant, we performed the Wilcoxon signed-rank test. The null hypothesis of statistically equivalent median values was rejected for statement and branch (with *p-values* 0.01392 and 0.04964, respectively), but not for function (with a *p-value* of 0.4014). Achieving a lower median value for the branch criterion was an unanticipated result as we would expect the scope-aided approach to be outperformed in all the cases when the set of all faults is considered. The results for function and statement were in accordance with our initial intuition.

Figure 4.7 displays the boxplots for each approach when considering the set of in-scope faults. The results achieved by the scope-aided approach improved with respect to the previous comparison considering the set of all faults and lower median values was achieved for the three coverage criteria. After a further evaluation, however, we noticed that the differences in the median values were statistically significant only for the branch criterion. For function and statement, the null hypothesis

of statistically equivalent medians could not be rejected. The resulting *p-values* for the Wilcoxon test were 0.05116, 0.1595, and 0.03909, for function, statement, and branch, respectively.

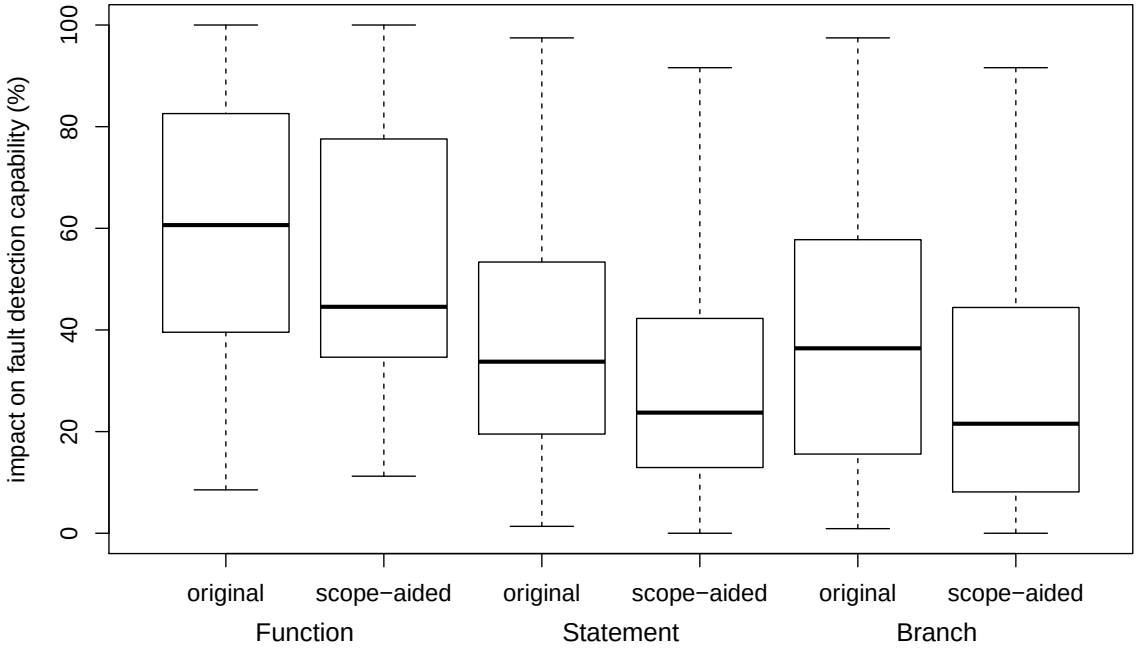


Figure 4.7: Impact on fault detection capability for the different coverage criteria when considering the set of *in-scope faults*

When looking from the perspective of the different subjects investigated, the results were very similar to the ones observed in our study with selection (Section 4.5.2): scope-aided approach had an average impact of 30.72% (against 33.80% generated by the traditional) and 36.16% (against 59.92% generated by the traditional) for grep and gzip, respectively. For sed, scope-aided generated an average impact of 47.62%, whereas the traditional approach generated an impact of 44.28%.

4.7 Discussion on the Costs and Benefits of the Approach

In sections 4.4, 4.5, and 4.6 we investigated the potential usefulness of scope-aided test prioritization, selection, and minimization. On the other hand, we also need to consider the possible costs of our proposed approach. While only data collected from real-world usage (which are not available) could provide conclusive evidence about whether such costs are sustainable and justified, in this section we present a preliminary assessment regarding the costs incurred by the adoption of the proposed approach.

In general, because the cost-benefit ratio incurred by the adoption of a given technique largely depends on the specific testing scenario in which the technique is going to be applied, deciding whether or not its adoption is worthwhile needs to be done on a case-by-case basis. For example, in the ideal case where the test suite is fully automated and one can analyze the test results within a negligible time, any prioritization technique, even the most efficient one, would be worthless in practical terms. The same could be said for selection and minimization techniques.

These techniques become attractive when the execution of the whole test suite is not doable due to resource constraints (e.g., computer-time, person-time, etc); or, for the specific case of prioritization, when the execution of the whole test suite would take so long that it is worth prioritizing the test cases in order to reveal potential faults faster and to maximize the benefits in the case that testing needs to be prematurely halted.

The literature offers countless variations of prioritization, selection, and minimization techniques. In all cases, these techniques start from a test suite T and derive a test suite T', which is a reordering of T in case of prioritization, and a subset of T in case of selection or minimization. Let us indicate C_{ORIG} as the cost of executing the original test suite T, and C_{POST} as the cost of testing after applying the technique. C_{ORIG} can be expressed as $C_{exec} + C_{act}$, and C_{POST} as $C_{tech} + C_{exec'} + C_{act'}$, where C_{tech} is the cost (e.g., effort, time) required for applying the technique (minimizing a test suite, for example); C_{exec} and $C_{exec'}$ are the cost required for running the original test suite T and the set T' resulting from the application of the technique, respectively; and finally C_{act} and $C_{act'}$ denote the cost of analyzing the test results and taking appropriate actions (fixing the bugs founds, for example) before and after applying the technique, respectively.

We recall that our approach is proposed as a boost of other existing prioritization, selection and minimization techniques. So, by applying a scope-aided (prioritization, selection or minimization) technique we obtain a test suite T'_s different from the test suite T' that would be obtained by the same technique without considering scope. As in previous sections we assessed the benefits of our approach against some state-of-the-art prioritization, selection and minimization approaches, consistently we will discuss augmented or reduced costs of scope-aided approaches with respect to the original (not scope-aided) ones.

With regards to the cost of applying the technique (C_{tech}), when compared to the original technique, our approach has an extra cost to identify the in-scope entities, which will depend on the method chosen for performing this task. We remind that our approach consists of two main steps: (1) constraints identification and (2) in-scope entities identification.

With respect to (1), our approach presupposes that the information regarding the specific reuse scope is available. In other words, we assume that developers know which functionalities are going to be reused and this information could be available informally in their minds or in some formal specification document. Some manual intervention may likely be required to express the reuse scope in a format that can

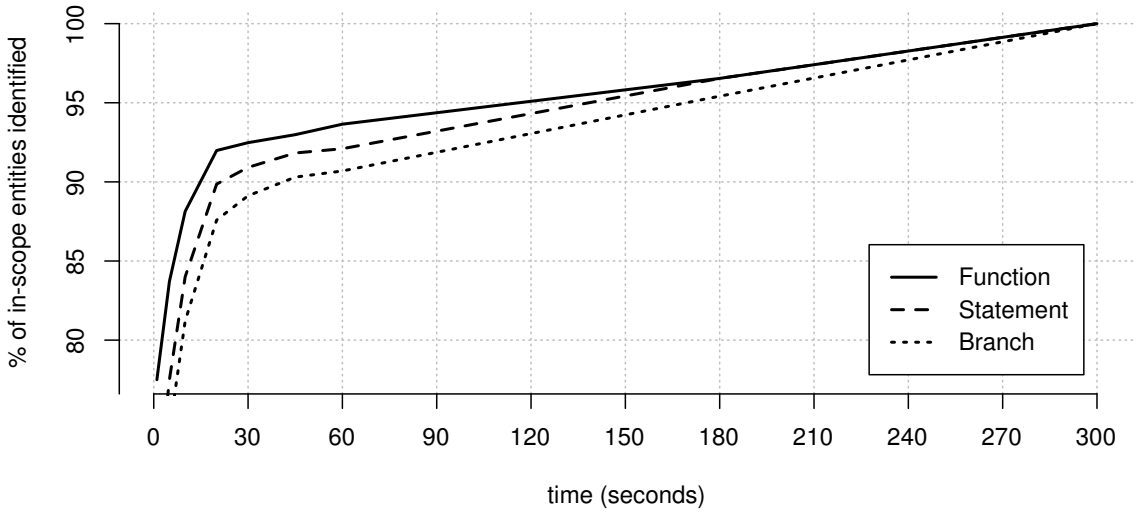
be used by the technology adopted for identifying the in-scope entities and this could incur costs that are hard to quantify in the general case. In our case we adopted DSE and this first step was as simple as expressing the reuse scope constraints into concrete parameters for the tool we adopted.

Regarding (2), the in-scope entities identification can then be done in automated way. As previously stated, in our studies we adopted DSE and we decided to allow the DSE tool to explore the source code of each subject for a maximum of 5 minutes, based on a previous research paper [20] that reported that this time budget was enough to allow KLEE to achieve high coverage exploration even for complex and environmentally-intensive programs.

We considered 5 minutes to be a reasonable amount of time for the purposes of our study, as this task is performed just once for each testing scope and the same results can then be used by all the techniques investigated (prioritization, selection, and minimization). However, if such time budget is not doable for some testing scenario this would not compromise the adoption of our approach as the vast majority of the in-scope entities can be identified after just a few seconds of DSE exploration. As we can see in Figure 4.8, on average, 90% of the in-scope functions and statements can be identified within 30 seconds. For the branch coverage criterion, 90% of the in-scope entities are identified at around 45 seconds.

Although DSE is largely adopted in research, it is known for limited scalability due to classic challenges such as the path explosion, for example [22]. Thus, for adopting our approach for very large software, if the limited scalability cannot be mitigated, a different technique for identifying the in-scope entities should be considered.

Figure 4.8: In-scope entities identification over a time period of 5 minutes



We now provide some discussion about the costs related to the individual testing techniques investigated in this work. For prioritization techniques one can consider roughly that $C_{exec} = C_{exec'}$ and $C_{act} = C_{act'}$, as prioritization per se does not involve a reduction of the test suite size, but only a modification of its execution ordering. Then we would have that with any prioritization technique (including scope-aided ones) $C_{POST} > C_{ORIG}$, because the former will have some additional C_{tech} associated. It is important to notice, though, that the real benefit provided by prioritization lies in the fact that faults can be revealed faster — which means that developers can start debugging activities as soon as possible — and the *de facto* metric to quantify the gains is the APFD_C.

As reported in Section 4.4, overall the scope-aided approach improved the APFD_C for all the subjects investigated, with the biggest improvement being an increase of 40.35% in the average APFD_C for gzip using the total approach. Besides that, when considering different fractions of a given test suite (75%, 50%, and 25%) scope-aided prioritization performed better than the not scope-aided one in the majority of the cases. In particular, for the 25% fraction, our approach defeated the traditional one in all the cases considered, even if to a small extent.

Regarding the cost of selection and minimization, in general we have that these aim at decreasing the overall cost of testing by reducing the size of test suite T'. This is of course true when the savings in terms of decreased $C_{exec'} < C_{exec}$ and $C_{act'} < C_{act}$ are higher than the cost of applying the technique C_{tech}. With our approach we saved, on average, 176 test cases for grep (22 tests in addition to those saved by the traditional approach); 190 test cases for gzip (11 extra test cases); and 355 tests for sed (14 test cases besides those saved by traditional). These results are displayed in percentage format in Table 4.7. The average figures for minimization (Table 4.8) were very similar and the maximum number of extra test cases saved were 19, 30 and 55 for gzip, sed and grep, respectively. Again, deciding whether the potential saving is worthwhile against the cost of the technique, will depend on the specific test environment and the cost incurred into executing and analyzing each test case.

4.8 Threats to Validity

In this section we present a summary of the potential threats to validity of our study, including:

- threats to internal validity: concern aspects of the study settings that could bias the observed results;
- threats to external validity: concern aspects of the study that may impact the generalizability of results; and
- threats to construct validity: concern confounding aspects by which what we observed is not truly due to the supposed cause.

4.8.1 Threats to Internal Validity

Testing scenarios representativeness: We defined the testing scopes for this study based on possible realistic uses of the subjects chosen. Real use of the subjects may include different scenarios that had not been considered in this study. We controlled this threat by carefully reading the subjects' documentation to understand well how they could be used before selecting the testing scopes for our study.

In-scope entities identification: As stated, we used KLEE for performing the symbolic execution of the subjects' code and identifying the set of in-scope entities. To deal with the path explosion, a classic challenge for symbolic execution [22], it is usually necessary to define a time budget within which the path exploration will occur. Because some of the KLEE's search heuristics for path-finding are random-based, if the exploration is halted by a timeout, the output is not deterministic (i.e., a different set of in-scope entities could be identified in different runs). To minimize this threat, we allowed KLEE to explore our subjects' code for 5 minutes expecting that the majority of the paths would had been explored within this time budget (in [20], the authors reported that some of the `GNU Coreutils` programs achieved high coverage exploration within three minutes).

SIR test suite coverage: The SIR test suites used for our investigations does not achieve full coverage of the study subjects. Different results could had been achieved if full coverage was provided, as the level of coverage may affect effectiveness, see, e.g., [64]. One possible way of controlling this threat would be adding more test cases to achieve different levels of coverage up to full. We performed a cost-benefit analysis and decided to use each subject as it is provided (i.e., not to introduce other test cases) with all of its artifacts, since it represents the *golden standard* for benchmarking purposes.

In-scope faults identification: The set of in-scope faults derived for computing the APFD_C and for evaluating the impact on the fault detection capability may not contain all the relevant faults that could be possibly revealed in a given testing scope. However, since we use the same fault matrix for both the original approaches and the scope-aided ones, we do not see how such threat could produce different impacts on different evaluations in systematic way, and thus influence the results on fault detection rates.

Prioritization evaluation: In this study we used APFD_C as the metric for evaluating the speed in which faults are revealed by a given test suite. However, APFD_C is not the only possible measure of rate of fault detection. Control for this threat can be achieved only by conducting additional studies using different metrics for evaluating the prioritization quality of the test suites used in our study.

Selection and Minimization evaluation: We adopted two metrics for evaluating the selection and minimization of the test suites in our study: (i) test suite reduction and (ii) impact on fault detection capability. However, there exist other metrics that could be adopted to evaluate the quality of the produced test suites. Control for this threat can be achieved only by conducting additional studies using different metrics for evaluating the selection and minimization quality.

4.8.2 Threats to External Validity

Subject representativeness: In this work we investigated the proposed scope-aided testing approaches on 17 variant versions of C programs. We acknowledge that the 17 versions belong to three subjects only. However it is important to notice that, in some cases, the differences between versions of a same subject were quite significant. Indeed, when considering the case of sed, for example, the development of v5 spans almost 5 years and the differences when compared with v4 are astounding — nearly every "major" function has been changed significantly. So, these could be considered as different programs. A similar case of fairly significant differences happened between versions v6 and v7 (as reported in the accompanying material from the SIR repository). Nevertheless, additional studies using a range of diversified subjects should be conducted for better representativeness. As explained in Section 4.3.1 the study settings imposed a set of requirements on the subjects that made it not easy to identify good candidates. Besides that, as stated before, our experiments were computationally intensive and very time-consuming.

Faults representativeness: In our study we considered mutant faults and subjects with real faults might yield different results. Control for this threat can be achieved only by conducting additional studies using subjects with real faults.

Prioritization strategies: We assessed scope-aided prioritization when used as a boost for coverage-, similarity-, and search-based prioritization strategies. Other prioritization techniques and more instances of the strategies already studied should be investigated before more general conclusions can be drawn.

Selection and Minimization strategies: Analogously to the previous threat, in this study we investigated only one selection (and one minimization) strategy. Other strategies should be investigated before more general conclusions can be drawn. Control for this threat can be achieved only by conducting further studies using different selection (and minimization) techniques.

4.8.3 Threats to Construct Validity

Study design: We introduced scope-aided testing approaches for prioritization, selection, and minimization that focus on giving priority to the detection on the

most relevant (in-scope) faults. In the study we wanted to evaluate the fault detection rate (for prioritization) and the impact on fault detection capability (for selection and minimization) of the original and the scope-aided approaches on the targeted subset of in-scope faults. We were conscious that constructing such a study could suffer of biased design, as we were using our testing approaches' goal (the in-scope faults) also into the evaluation metrics ($APFD_C$ and impact on fault detection capability). The only way to prevent this threat would have been to use study subjects with failure reports from real world usage, but we were not able to find such subjects. To mitigate the threat, we used large suites of random test cases, different from the baseline test suite (the one that is later prioritized, selected and minimized), to identify the in-scope faults.

4.9 On the Adoption of Relevant Coverage for Test Adequacy

In this section we report preliminary results of an ongoing exploratory study we are conducting to investigate the usefulness of the proposed relevant coverage as an adequacy criterion. Notice that the order of presentation of the chapters in this thesis does not reflect the chronological order in which our investigations were conducted. These results are reported here because, even if not complete, they are related to the relevant coverage.

The experimental materials we are using, including study subjects, testing scopes, mutants, and other artifacts, are the same adopted for the exploratory study already reported in this chapter, in Section 4.3. Some of the tasks and procedures, however, are slightly different and they are presented next in Section 4.9.1. In our exploratory study we are focusing on the following research question:

Does relevant coverage provide a good adequacy criterion (stopping rule) for the context of software reuse?

4.9.1 Tasks and Procedures

We considered three types of entities: function, statement, and branch, which correspondingly identify three coverage criteria. Then, for each combination of subject, version, testing scope, and coverage criteria we performed the following tasks:

1. We derived 100 test suites by randomly selecting test cases from the subjects' suite available from SIR. For each test suite, we stopped when: (i) 100% coverage was achieved for both traditional and relevant coverage; or (ii) the derived test suite reached 100 test cases.

2. After each test case is selected, we measured the traditional and relevant coverage achieved. Then, at different coverage levels (10%, 20%, ..., 90%, 100%) we recorded information about:

(a) the number of test cases in the relevant test suite;

(b) the number of test cases in the traditional test suite;

(c) the impact on fault detection capability for the relevant test suite

4.9.2 Test Suite Size

Figure 4.9 displays the average test suite size measured at different coverage levels. The results are grouped by the different subjects and coverage criteria considered. For each plot, the x-axis represents the target coverage level, while the y-axis displays the number of test cases in the derived test suites. In this figure, traditional coverage and relevant coverage are represented by the continuous line and the dashed line, respectively.

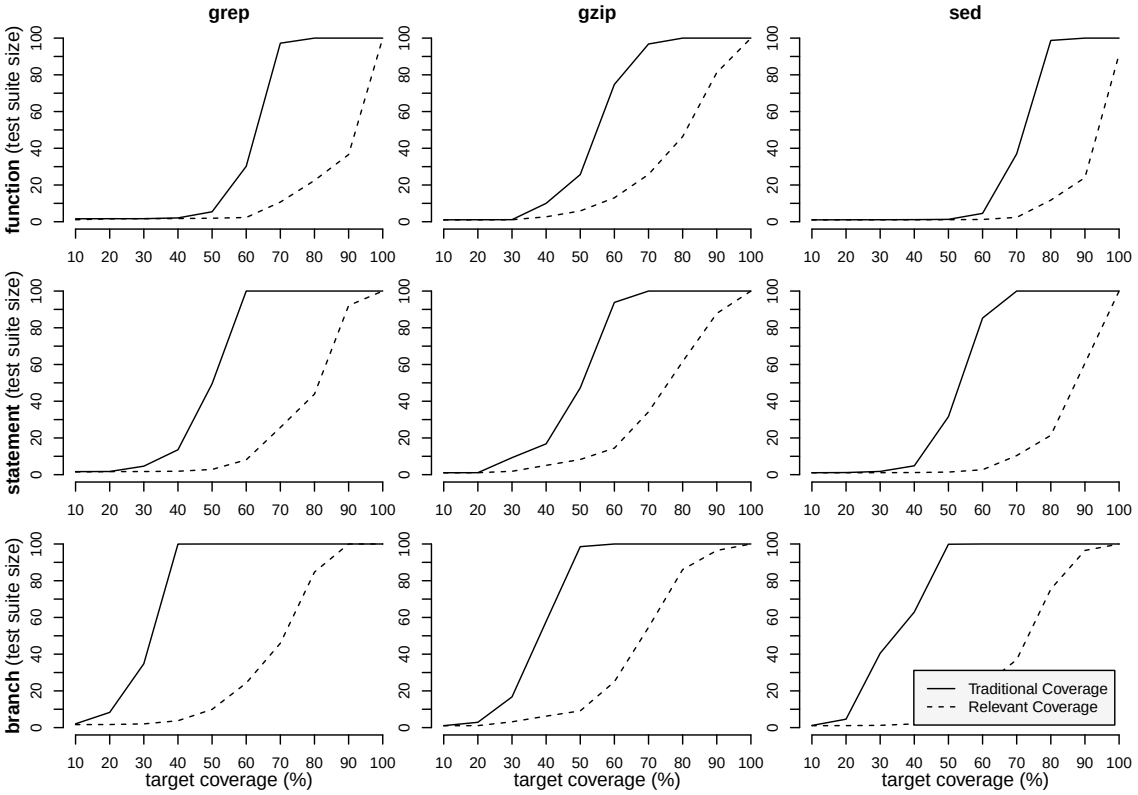


Figure 4.9: Average test suite size at different coverage levels

From the graphs, the main observation is the fact that, in general, relevant coverage criteria requires much less test cases than traditional coverage. The only exception happened for grep when targeting 30% of function coverage. In that particular case, the target traditional coverage was achieved after 1.66 test cases, while the same level of relevant coverage was achieved after 1.67 test cases.

Another observation is the fact that 100% coverage, either relevant or traditional, could not be achieved for the vast majority of the cases. For sed, when targeting

function coverage, 100% relevant coverage was achieved after an average of 91.2 test cases. For all the other cases we can see that the two curves reached 100 test cases (y-axis) at some point.

4.9.3 Impact on Fault Detection Capability

Table 4.9 shows the average impact on fault detection capability of the relevant test suite. It also displays the average size of the test suites targeting both traditional and relevant coverage criteria (columns $|trad|$ and $|rel|$, respectively). The impact on fault detection is calculated according to Equation 4.4 introduced in Section 4.3.3. In this table, we show only the results for the coverage levels that are more commonly targeted, i.e., 70%, 80% and 90%. Results for 100% are not displayed because, as explained before, it was not reached in the vast majority of the cases.

In all the cases observed, the relevant test suite suffered some impact on its fault detection capability when contrasted with the faults revealed by the traditional one. This result was expected as the relevant test suite usually contains a lower number of test cases. If we consider, for example, the results for function when targeting 90% coverage, the relevant test suite had an impact on its fault detection capability of 18.19%, but it contained a little bit less than half the number of test cases in the traditional test suite (an average reduction of 52% in the test suite size). An interesting observation is the tendency of reduction in the impact when more fine-grained criteria are considered: for a given target coverage, the impact when considering statement coverage was smaller than that for function coverage; and the impact for branch was smaller than that when considering statement coverage.

Table 4.9: Average impact on fault detection capability at different coverage levels

Target Coverage	**Function**			**Statement**			**Branch**		
	$\|trad\|$	$\|rel\|$	*impact*	$\|trad\|$	$\|rel\|$	*impact*	$\|trad\|$	$\|rel\|$	*impact*
70%	78	13	44.87%	100	27	33.91%	100	49	20.60%
80%	100	27	38.62%	100	45	21.67%	100	83	4.60%
90%	100	48	18.19%	100	81	5.79%	100	98	0.35%
Average:	93	29	33.89%	100	51	20.46%	100	77	8.52%

We acknowledge that the analysis of these two metrics alone (i.e., test suite size and impact on fault detection capability), is not sufficient for drawing general conclusions. Currently, we are designing a new exploratory study in which we will investigate the correlation between the growth in the coverage measure and its impact on fault detection. This study will be similar to the one reported in Chapter 6 (Section 6.3) in which we investigated the correlation between the coverage achieved and the probability that the next test case derived according to the user's profile would fail. By performing a more rigorous exploratory study, and backed by statistical analysis, we expect to be able to provide more general results.

4.10 Related Work

In this chapter we have introduced a coverage criterion tailored for the context of software reuse and we demonstrated how it could be used for supporting the proposed scope-aided testing approach to boost test prioritization, selection and minimization of reused software. The work presented here is mainly related with the topics of coverage testing; prioritization, selection and minimization for regression testing; and testing of reused code; for which a preliminary discussion was provided in Sections 2.1, 2.2, and 2.3, respectively.

The newly introduced coverage criterion, the Relevant Coverage, is inspired by the idea of relative coverage. In [13], Bartolini et al. claimed that traditional coverage testing should be revised to deal with service-oriented systems, and introduced the notion of relative coverage in which the set of covered entities was measured against a customized set of entities that could vary from a user to another. Relative coverage was later used also in [33]. In this work, the authors proposed an approach in which testable services (services instrumented to provide their clients with coverage information) were provided along with test metadata to help their testers to get a higher coverage. In both [13] and [33], however, the list of in-scope entities needed to be manually defined by the user, whereas here we provide an approach for deriving them automatically.

Our work builds on a huge literature of white-box approaches for test prioritization, selection and minimization, as presented in a comprehensive survey conducted by Yoo and Harman in 2012 [137]. At a later date, Catal and Mishra [23] provided a systematic mapping study focusing on test case prioritization only. Then, the never-ending interest in those topics led to a myriad of recent work related to test case prioritization (e.g., [59, 5, 60, 83, 122], just to mention a few); test case selection (e.g., [95, 108, 118, 3]); and test suite minimization (see, e.g., [80, 67, 44, 128]). However, to the best of our knowledge, techniques and tools in the literature are mostly conceived for regression testing purpose, and there exists no previous work that expressly targets test prioritization, selection and minimization when an existing test suite is used to test a software component or a piece of code that is reused in a new context. To confirm such statement, we have made a systematic search within both ACM DL and IEEExplore repositories, looking for any paper that included in the abstract both terms “reuse” and “test” (or “testing”), and any of the three words: “prioritization”, “selection” or “minimization”. We then screened all papers returned by the search, and in fact among them we could not find any relevant reference[7].

Although in practice the same techniques proposed for regression testing could be applied for reuse, conceptually the two testing activities have quite different purposes. In both cases we retest a software using a set of test cases from an existing test suite. However, in regression we do this to validate changes to the

[7]The systematic search was conducted in April, 2016.

software under test, and in reuse to validate the same software being used in a different context. Therefore it is natural to target test cases differently in the two cases. An approach like ours that specifically targets those test cases satisfying the new domain constraints may make sense.

In test suite prioritization test cases can be re-ordered based on various criteria [29, 53]. So far criteria used by other authors include test case execution time, history of failure detection, fault localization costs, and others, but to the best of our knowledge no work exists that relates program entities to their relevance in a reuse context and uses such information to re-order them, as we do. Our approach prioritizes white-box test cases based on possible constraints on the input domain in order to focus on more relevant faults in a reuse scope. In a recent work, Henard et al. [59] provided a comparison between white-box and black-box test prioritization. The authors found little difference between black-box and white-box performance with a maximum difference of 4% in the fault detection rate. Here we focused only on white-box prioritization techniques, but our approach could also be adopted for black-box prioritization (for example, by applying the notions of testing scope, in-scope entities, and out-of-scope entities to the list of requirements to be tested).

Achieving adequate coverage may require a high number of test cases, and researchers have proposed several approaches for test selection and minimization, e.g. [72, 41]. In a recent work, Mondal et al. [95] compared two different heuristics for test case selection (one based on code coverage information and the other one based on test case diversity), and they concluded that neither of the two techniques completely dominates the other. The authors, then, proposed a multi-objective test case selection approach that combines both code coverage information and test case diversity, and they reported improvements in the fault detection rates ranging from 10% to 16%. Concerning test suite minimization, Lin et al. [80] proposed to reduce the cost of regression testing by adopting cost-aware algorithms. The authors evaluated the proposed approach against traditional minimization techniques, and they reported a decrease in the costs of the minimized test suites that varied from 8.10% to 46.57%. To some extent, the cited works [95, 80] could be said to be similar to ours as both aim at improving traditional techniques by combining coverage data with additional information. In our case, coverage data is enriched with scope information to boost traditional approaches. Scope-aided approaches for test selection and minimization during reuse testing aim at reducing the size of a test suite based on the principle to retain those test cases covering the code entities that really matter, and discard test cases covering entities that are outside scope.

While software reuse may involve reuse of software artifacts and documentation beyond mere code reuse, our work is related to reuse of code, both in systematic [35] or pragmatic forms [34]. Indeed, depending on the extent and formality under which code is reused, testing difficulties and approaches vary. The research of testing in software reuse has followed two main directions: product-lines testing and component-based testing. As surveyed in [123, 34], various testing frameworks have been proposed for product lines. The main challenge in product line testing is to

account for variability among products, and not among different usage contexts or scopes. Therefore, we consider our approach not closely related to product line testing. In component-based testing [42] researchers have investigated both approaches for testing and certifying one component built for reuse, and approaches for testing the integration of a component into a system [65]. Our research is more related to the second case, in that our approach can be used for testing the integration of a component into a system, assuming that the component code is available, as is generally the case in internal reuse. A component developer following the approach proposed in [106], could incorporate in a component metadata relevant information to facilitate the derivation of possible usage scenarios in component reuses. In [138] a framework is presented that supports prioritization of compatibility testing in component integration: the aim is close to our goal, however prioritization focuses on evolution and regression, and not on the usage context.

4.11 Concluding Remarks

In front of a huge literature on challenges and approaches on the one side to support and promote software reuse, and on the other side to improve software testing cost-effectiveness, the very problem of testing reused code has surprisingly received scant attention. In particular, while coverage measures are widely used in software development, appropriate coverage testing tools for code that is reused are lacking [124]. There exist many regression testing techniques and tools that address the retesting of software after maintenance in order to ascertain that the modifications have not produced undesired effects. As such, these approaches naturally focus on those test cases that exercise the software functionalities or code entities impacted by the changes. When testing a reused code (be it changed or unchanged) into a new context, the main goal is to test the ways in which the software is invoked from the new context. Quoting Weyuker [131], *when developing a component for a particular project or application, ... testers usually have some information or intuition about how the software will be used and therefore emphasize, at least informally, testing of what they believe to be its central or critical portions. These priorities will likely change, however, if it is decided to incorporate the component into a different software system.*

The approach of scope-aided testing we propose here entails exactly leveraging the information or intuition about how the software will be used in each new context or scope to redirect the focus of existing white-box prioritization, selection and minimization, and thus make them more cost-effective for reuse testing.

In brief our scope-aided testing approach requires developers to make explicit any constraints limiting the reuse input domain. The identification of reuse constraints is the first basic step, and the only one which obviously requires human intervention. After the scope constraints are given, the approach can be completely

automated, and in this paper we have leveraged in particular dynamic symbolic execution techniques.

We use such constraints first to identify in-scope entities (as described in Section 4.2), and then to adapt an existing technique to privilege these in-scope entities when picking the next test case to execute. We have shown how to do so considering some existing prioritization (Section 4.4), selection (Section 4.5) and minimization (Section 4.6) approaches, however the basic idea is general and other techniques could be adapted as well.

The results from our empirical evaluation are encouraging: although scope-aided did not win consistently in all studies, when looking in particular at those cases in which only a limited number of test cases can be executed, scope-aided clearly outperformed not-scope aided approaches.

When only 25% of the test cases are executed (see Table 4.4), for example, scope-aided prioritization defeated the traditional approaches for all the coverage criteria considered by finding in-scope faults faster. When applied to minimize test suites, scope-aided consistently produced lower impact in terms of reducing the in-scope fault detection capability (see Figure 4.7). In this chapter we have reported summary results from our empirical studies. More detailed data can be found on-line at `http://labsedc.isti.cnr.it/tools/scope-aided-testing`.

Concerning the future, further experimentation is needed to know more about the usefulness of scope-aided approaches to reuse testing. We would like to address other subjects and other test techniques, as well as to study other possible methods to identify in-scope entities. We aim at increasing the data reported in the above mentioned page as more subject are studied.

The approach proposed in this work can only be applied in those reuse contexts in which the source code of the reused software is available. However, the studies conducted here did not take into account the impact of changes in the reused code to the scope-aided approach. Our intuition is that our approach can still be applicable and useful, but we would need to run additional experiments in a setting that explicitly models code changes and assess its cost-effectiveness in order to confirm (or not) this intuition.

Besides, we envisage that the same idea of scope could be applied to black-box reuse considering other types of test requirements, for example the specification of functional requirements. In the same way that we distinguished here between in-scope and out-of-scope code entities, we could identify in-scope requirements and adapt the functional testing of a reused code accordingly. Similarly, one could think of prioritizing or selecting test cases addressing configuration testing, by privileging those configurations that are more relevant in a reuse context.

5

Customizing Coverage Criteria Based on Coverage Data from Similar Users

In the previous chapter we presented a coverage criterion that customizes coverage measures according to the testing context in which a code is being reused. We called it "Relevant Coverage" and we discussed how it could be particularly useful for boosting traditional approaches for test case prioritization, selection and minimization.

Relevant coverage, as presented in the previous chapter, presupposes that the source code is available. In this chapter we present how the relative coverage idea could still be useful for improving testing cost-effectiveness even in the absence of source code. We motivate the proposed approach by referring to the motivating example provided in Section 3.2. The issues illustrated in the example 3.2 are similar to the ones discussed in the previous chapter because the scenario portrayed can also be seen as a reuse scenario: services are *reused* by developers for composing their own, possibly bigger, services. However, the SOA paradigm poses additional complications on top of the ones existing for traditional reuse as the source code of the reused services is not available. Here, in an attempt to provide relevant coverage metrics for the service consumers, we introduce a new coverage criterion that customizes coverage information in a given context based on coverage data collected from similar users. We call it the "Social Coverage".

We start by describing our approach to calculate social coverage in Section 5.1. Next, in Section 5.2, we present the results of the exploratory study we conducted to evaluate its potential. Related work is presented in Section 5.3. Finally, our concluding remarks for this chapter are presented in Section 5.4.

5.1 Social Coverage

To overcome the limitations of traditional coverage illustrated in the motivating example 3.2, we propose the social coverage: a coverage criterion that customizes coverage information in a given context based on coverage data collected from similar users.

Social coverage is calculated in a analogous way to that of relevant coverage, as displayed in Equation 5.1. Indeed, the only difference is that, for social coverage, the in-scope entities are those that are covered by similar users, i.e., those in the *social scope*.

$$\textbf{Social coverage} = \frac{\textit{number of covered entities}}{\textit{number of entities in the social scope}} \cdot 100(\%) \qquad (5.1)$$

Recall that for relevant coverage we use dynamic symbolic execution for identifying the set of entities that would be relevant to a given reuse context under the input domain constraints provided. When the source code is not available, however, such approach is not be applicable. Social coverage, thus, relies on coverage information collected from similar pairs. More details are provided in the next sections.

5.1.1 Approach

As for any coverage criterion, social coverage measurement presupposes that the code is instrumented so to allow the identification of the entities exercised by customers (they could be humans or other software). For a target tester (i.e., the tester who is interested in receiving coverage information), social coverage measurement can then be summarized in the following steps:

1. **Data collection**: the entities exercised by the target tester are monitored. In addition, for social coverage testing it is assumed that usage information, including the list of entities exercised, are being routinely collected for all customers using the system under test.

2. **Similarity calculation**: the similarity between the target tester and all other system customers is calculated.

3. **Identification of *in-scope entities***: the list of in-scope entities is obtained by combining information about the entities exercised so far by the tester and the ones exercised by *similar* system customers.

4. **Coverage measurement**: social coverage is calculated and the results are provided to the target tester.

5. **Coverage increase**: a list of possibly relevant entities that have not been exercised by the target tester is suggested.

To better explain the approach, in the next section we apply it to the motivating scenario depicted in Section 3.2.

5.1.2 Illustration

Figure 5.1 depicts a service provider S implementing a Travel Reservation System (TRS), whereas Figure 5.2 illustrates a new service consumer, service N, that invokes operations from the service provider S. Figures 5.1 and 5.2 are the same displayed in Chapter 3. We display them again here for the sake of improving readability.

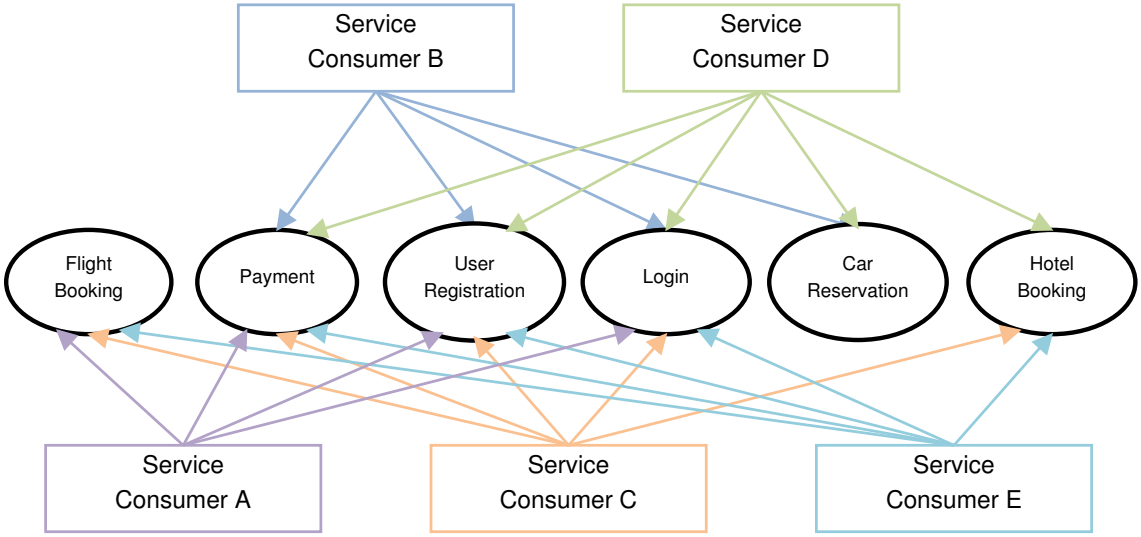


Figure 5.1: Travel Reservation System implemented by service provider S.

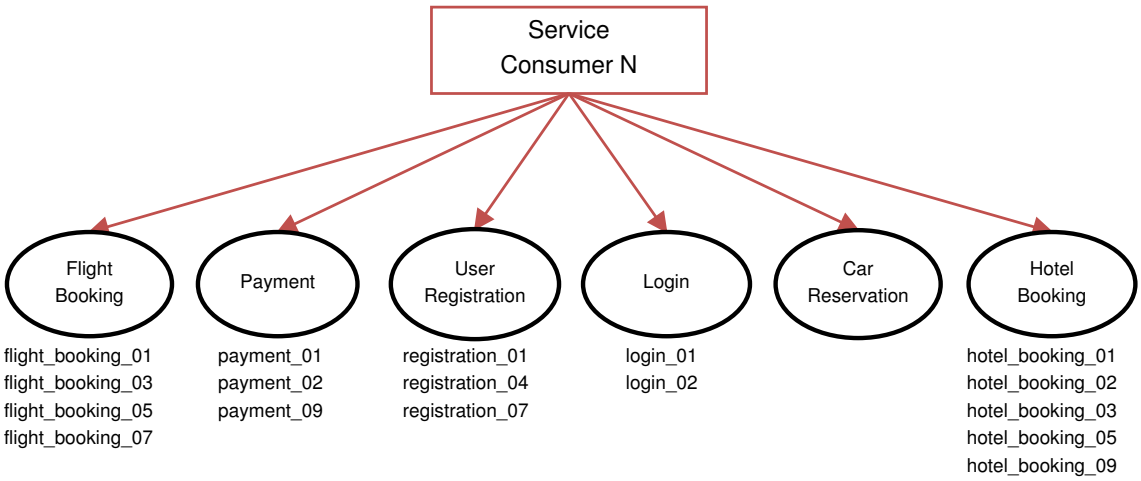


Figure 5.2: Service consumer N.

Owning the instrumented code, the service provider S can get detailed information about the operations used by each of its service consumers. As the first step of our approach, using this information the service provider S can calculate the similarity between the service N and all the other service consumers to find the most similar services.

Many similarity measures could be adopted to find similar service consumers: Jaccard similarity coefficient [58], Pearson Correlation [47], and Euclidean Distance [121] are just a few examples. For this illustrative example and for the exploratory study presented next, we used the Jaccard similarity coefficient. This decision was taken based on the results achieved by Hemmati and Briand [58]. In this work, the authors compared the cost of different similarity functions both in terms of computational complexity and the actual time required for the similarity calculation, when applied to the task of test case selection, and Jaccard was shown to be the most cost-effective similarity measure.

Table 5.1 reports the Jaccard similarity coefficient (presented as percentage) between service N and the other service consumers. The table shows that, based on the operations used by each client, service consumer E is the most similar service when compared to service N with a similarity degree of 88%; service consumer C is the second most similar service with a similarity degree of 70%; and so on.

Having calculated the similarities, the next step is identifying what are the in-scope entities for service consumer N, that is, the operations that might be of interest to service consumer N and should be considered when calculating coverage (the denominator of Equation 5.2). For this task, it is important to define a similarity threshold to guarantee that only the most similar service consumers are considered when defining the set of targeted entities.

To continue with our example, let us assume that we are interested in analysing only the service consumers whose similarity with N is at least 70%. In this case, services A, B, and D would be discarded. As we are using the Jaccard similarity coefficient in this example, a simple way to define the entities targeted by N would be considering the union (or the intersection) of the operations invoked by the most similar service consumers. The adoption of collaborative filtering technique [116], on the other hand, would be more powerful as it would allow us not only to identify the set of entities exercised by similar clients, but also to estimate the probability that the service N is interested in a given entity. Such improvements will be deeper investigated in future work.

Using the union of the operations exercised by similar service consumers, the set of targeted entities for N is composed by 23 operations: all operations from service consumer C plus `hotel_booking_09` (invoked by E). We can now calculate the social coverage according to Equation 5.2:

$$\text{Social coverage} = \frac{\textit{number of covered entities}}{\textit{number of entities in the social scope}} \Rightarrow \frac{17}{23} \approx 74\% \qquad (5.2)$$

In comparison with traditional coverage calculated according to Equation 1.1, this is a more meaningful measure: the point is not in merely achieving a higher score, but in having a more realistic estimate of what could be achieved by augmenting the test suite.

Moreover, service S can provide a list of possibly relevant operations that have not been invoked by N (see Table 5.2). Since the list shown in Table 5.2 has been

Table 5.1: Similarity calculation for service consumer N

Service Consumer	Covered Entities	Similarity with N
Service A	flight_booking_01, flight_booking_02, flight_booking_04, flight_booking_07, flight_booking_08, payment_01, payment_02, payment_09, registration_01, registration_04, registration_07, login_01, login_02	50%
Service B	payment_01, payment_02, payment_03, payment_09, registration_01, registration_04, registration_05, registration_07, login_01, login_02, car_reservation_02, car_reservation_03, car_reservation_05, car_reservation_07	35%
Service C	flight_booking_01, flight_booking_03, flight_booking_05, flight_booking_07, payment_01, payment_02, payment_03, payment_04, payment_05, payment_09, registration_01, registration_02, registration_03, registration_04, registration_05, registration_07, login_01, login_02, hotel_booking_01, hotel_booking_02, hotel_booking_03, hotel_booking_05	70%
Service D	payment_01, payment_02, payment_09, registration_01, registration_04, registration_07, login_01, login_02, car_reservation_03, car_reservation_04, car_reservation_07, hotel_booking_01, hotel_booking_02, hotel_booking_05, hotel_booking_08	52%
Service E	flight_booking_01, flight_booking_03, flight_booking_07, payment_01, payment_02, payment_09, registration_01, registration_04, registration_07, login_01, login_02, hotel_booking_01, hotel_booking_03, hotel_booking_05, hotel_booking_09	88%

Table 5.2: Possibly relevant operations that have not been invoked by N

Un-tested Operations
payment_03
payment_04
payment_05
registration_02
registration_03
registration_05

customized for N according to the behavior of similar services, it can be much more helpful in guiding testers to decide whether or not they need to increase the coverage of their test suites than simply providing the full list of operations available in the system. In the case the tester decides to create new test cases to cover previously un-tested operations, steps 1 to 5 of our approach are repeated and social coverage information is updated according to the new data available.

5.2 Exploratory Study

In this section we share our experience in instantiating the social coverage metric in the context of a real world system. In this exploratory study, we focus on the following research question:

> **RQ1:** *To what extent is our approach able to accurately predict the operations that would be relevant to a given user based on the analysis of the operations invoked by similar users?*

5.2.1 Study Setup

Our study was carried out in the context of gCube, a Service Oriented Infrastructure enabling software, supporting the definition of Virtual Research Environments (VREs). A VRE is a flexible and agile application development model based on the notion of Software as a Service (SaaS), in which components may be bound instantly, just for the time they are needed, and then the binding may be discarded. gCube has been implemented with the support of the European Commission in the context of a series of projects [43].

gCube was apt for experimenting our social coverage approach, since it is equipped with a monitoring tool based on a messaging system that logs, among other things, the usage of some of the available services accessed through a web portal. In particular, the log from the D4Science infrastructure [6], enabled by gCube, was used to carry out our study.

Whenever a portal operation is called, the following information are stored: the operation called, the user that made the call, the date and time, the calling scope,

and the message exchanged on that call. These data are made available through a consumer library. Such library allows the user to retrieve database content, exposes facilities to parse results as JSON objects, and aggregate information for the creation of statistics.

5.2.2 Tasks and Procedures

The steps below summarize the procedure followed in this study:

1. Collect usage data from D4Science portal using the consumer library API
2. Cleanup the data collected by removing fake users (used for testing purposes), portal administrators, and users with very few operations invoked
3. For each remaining user:
   (a) Split the data generated by that user into *training* and *testing* sets
   (b) Use the data available in the training set to predict what entities would be relevant to that user
   (c) Use the information from the testing set to compute the *precision* and *recall* of the predictions made

5.2.3 Preliminary Results

We collected usage data (step 1) from November 2009 to November 2013, which resulted in 94.768 invocations logged from a total of 484 portal users. In the cleaning up phase (step 2), we noticed that many users had very little information logged. For example, about half of the users had only one invocation logged. When working with predictions/recommendations, such situations could be affected by the cold-start problem [76], the very well-known issue that a given system cannot draw any inferences for users about whom it does not have sufficient information.

Defining the minimum amount of information required for obtaining good predictions/recommendations is a hard task. This is an acknowledged problem in the recommender systems literature [26, 112]. The authors of [26] investigated the effects of profile length for an explicit and rating-based elicitation strategy (in the movie recommendation domain) and suggested that the optimal number of ratings is more likely to be 10. Even though in our approach the data is collected implicitly (rather than explicitly), based on the findings of [26] we decided to remove all the users with less than 10 invocations logged to make sure that we would have a minimum of information available to split the data between training and testing sets. After the cleaning up phase, data from 159 users remained for our study.

For the step 3a, we used the first T invocations (under varying values for T) of a given user as the training set and the remaining ones as the testing set. To

Table 5.3: Precision and Recall (training set = 10 invocations)

User	# Similar Users	Predicted Entities	List of Actually Covered Entities	Precision	Recall
...	...	...	...	...	...
U_025	5	ga_i_loginrecord, hlrecord, loginrecord	ga_i_hlrecord, ga_i_loginrecord, hlrecord, loginrecord	100%	75%
U_026	2	loginrecord, ebob_sl_loginrecord hlrecord, ebob_sl_hlrecord, loginrecord	loginrecord, ebob_sl_loginrecord hlrecord, ebob_sl_hlrecord	80%	100%
U_027	10	hlrecord, loginrecord	ga_t_loginrecord, hlrecord, loginrecord	100%	66%
U_028	11	ds_dvre_loginrecord ebob_loginrecord, ebob_sl_hlrecord, ebob_sl_loginrecord hlrecord, loginrecord	ebob_loginrecord, ebob_sl_hlrecord, ebob_sl_loginrecord hlrecord, loginrecord	83%	100%
U_029	51	hlrecord, loginrecord	hlrecord, loginrecord	100%	100%
...	...	...	...	...	...
			Overall Average:	**99%**	**72%**

answer RQ1, after identifying the possibly relevant entities for a given user (step 3b), we evaluate the quality of the predictions made using two well-established metrics, namely precision and recall (step 3c). Precision measures how often the approach makes an appropriate prediction. In our context, an appropriate prediction is achieved for a given user when our algorithm suggests an entity that is, in fact, covered in the future. Recall measures how many of the operations invoked by a given user are actually predicted by the algorithm. Table 5.3 shows, for example, the values of precision and recall achieved with T=10. For the sake of space, the table shows only an excerpt of the results (the original table contains a total of 63 users). To preserve confidentiality, the data in the table have been anonymized by removing user names and by changing the actual operation names.

We observed the behavior of our approach when using different configurations. We repeated the study using different training set sizes (10, 20, 30, 40, and 50 invocations) and different values for the Jaccard similarity coefficient (0.50, 0.66, and

0.75). The training set size defines the amount of information (in our case, number of invocations) used to create the user profile. The Jaccard similarity coefficient, on the other hand, influences the decision of whether two given users are similar or not. For example, when using Jaccard similarity coefficient equal to 0.50, two users A and B are considered to be similar if the intersection of the operations invoked by them is at least 50% of the union of the operations invoked by each user individually.

Table 5.4 shows the overall precision and recall achieved when using different values for the Jaccard similarity coefficient. As we can see, the best results for both precision and recall have been achieved when using similarity equal to 0.75. A possible explanation for this result is the fact that the algorithm is more conservative when making predictions and, consequently, the predictions are more accurate. A trade-off of using high values for the Jaccard similarity coefficient, however, is that less predictions are made. As we can see in Table 5.4, the amount of users that received recommendations (predictions) varied between 23 and 63, depending on the different sizes of training sets (10, 20, 30, 40, and 50). Thus, it is important to find a balance between the number of predictions made and the accuracy that is appropriate for the environment in which the approach is being applied.

Table 5.4: Precision and Recall achieved for different values of Jaccard Similarity Coefficient

Jaccard Sim. Coefficient	User Span	Overall Avg. Precision	Overall Avg. Recall
0.50	49 to 100	73%	67%
0.66	38 to 88	78%	68%
0.75	23 to 63	97%	75%

Table 5.5 shows detailed information about the values of precision and recall achieved for the different training set sizes when using Jaccard similarity coefficient equal to 0.75.

Table 5.5: Precision and Recall achieved when using training sets with different sizes (similarity = 0.75)

Training Set	# Predictions made	Avg. Precision	Avg. Recall
10 invocations	63	99%	72%
20 invocations	44	98%	77%
30 invocations	36	96%	75%
40 invocations	28	97%	73%
50 invocations	23	96%	77%

Although the reported results are preliminary and cannot be generalized to other systems, they are promising and encourage us to perform further studies.

5.3 Related Work

In this chapter we have introduced a coverage criterion, the social coverage, for customized test adequacy and selection. It is proposed for the context of software reuse as an alternative to the relevant coverage (introduced in Chapter 4) when the source code is not available. The work presented here is mainly related with coverage testing and the notion of relative coverage. Both topics have already been anticipated either in Section 2.1 or in the related work section in Chapter 4. For the sake of conciseness, we do not cover these topics again. Instead, we ask the reader to refer to the pointed sections for more details. Additional related work include research from the topic of *collaborative testing* as presented next.

Various proposals are advocating to leverage information that come from the community of users in a collaborative testing approach [9, 126, 125]. In [9], for example, the authors proposed a test-broker architecture in which all the stakeholders within the composite web service can contribute to improve the testing of the services. Besides supporting the submission, indexing, and querying of test artifacts such as test cases, defect reports and evaluations, the test broker can also provide the services for the test generation, test coordination, and distributed testing services.

Tsai et al. [125] suggest that Collaborative Verification and Validation (CV&V) should be used as the testing paradigm for web service testing instead of Independent Verification and Validation (IV&V). The proposed approach can be used to rank the fault detection capability of test scripts and to establish the oracle for most test inputs.

Zheng and Lyu [140] proposed an approach in which past failure data of similar neighbors is used to predict the reliability of a given web service. This is very similar to our work as we also use past data of similar users to provide coverage information to a given tester. The main difference when compared to our approach is that, while in [140] the reliability of web services is predicted without providing further support for the testing activities, in our work we attempt to extrapolate the topic of coverage testing by customizing coverage information for a tester based on what operations have been invoked by similar users and by suggesting possibly relevant entities that have not been tried yet.

5.4 Concluding Remarks

In this chapter we have introduced Social Coverage, a coverage criterion that customizes coverage information for a tester based on coverage data collected from similar users. The proposed approach can mitigate the limitations of traditional coverage and can provide meaningful coverage results to testers, guiding them towards increasing test coverage in the areas that are relevant to them. We have hereby defined the approach, implemented an instance of it, and performed a preliminary study to evaluate its potential.

The results achieved in our preliminary investigations suggest that, in similar environments, social coverage can provide a better support to testers than traditional coverage. In our exploratory study, our approach achieved precision rates ranging from 73% to 97% (depending on the Jaccard similarity coefficient adopted) and recall ranging from 67% to 75%. However, thorough empirical studies still need to be conducted to evaluate whether such precision and recall levels are satisfactory to allow social coverage to be used for test adequacy purposes. Should they be considered satisfactory, then one practical implication of this is that computing similarity between users might be a cost-effective way of identifying the set of in-scope entities for a given testing context.

In the future, we plan to extend this work in several ways. First, we plan to adopt different techniques to find similar users (e.g., Pearson Correlation, Euclidean Distance, etc) and to recommend entities. This will allow us to objectively compare the different techniques is terms of the quality of the recommendations achieved, the computational cost, and the actual time required for the similarity calculation. Second, while in this work we studied the capability to predict a user/service coverage based on similar users/services, we plan to investigate whether social coverage can actually improve the testing process cost-effectiveness ratio. Third, we plan to investigate the usefulness of social coverage when considering different adequacy criteria such as statement coverage and branch coverage, for example. Finally, we plan to perform further studies to obtain a better understanding of the cost, performance, and potential benefits of adopting the social coverage adequacy criterion.

6

Introducing a Coverage Criterion for Operational Profile Based Testing

In the previous chapters we introduced two coverage criteria as an attempt to overcome the limitations of traditional coverage when applied to some specific contexts. More precisely, in Chapter 4 we presented a coverage criterion tailored for software reuse, and in Chapter 5 we proposed a coverage criterion that customizes coverage information in a given context based on coverage data collected from similar users. In both cases, we classify each entity of the system under testing as either an in-scope entity or an out-of-scope entity.

In this chapter we go one step further as we propose an approach that not only classifies the entities as in-scope or out-of-scope, but also assigns weights to the in-scope entities according to their expected usage in the system under testing. We introduce a new coverage criterion called "Operational Coverage" that was conceived with the motivation example 3.3 in mind and is customized to the usage profile of the entities to be covered.

We start by describing our approach in Section 6.1. Then, in Section 6.2, we present some details about the exploratory study we conducted to investigate the adoption of operational coverage for test adequacy and selection for operational profile based testing. Next, the results observed in our studies are reported in Sections 6.3 and 6.4. Discussion is presented in Section 6.5, followed by the threats to the validity of our study, reported in Section 6.6. Finally, related work is presented in Section 6.7 and our concluding remarks for this chapter are presented in Section 6.8.

6.1 Operational Coverage

Our proposed approach is meant to provide a practical stopping rule as well as a selection criterion for operational profile based testing. An operational profile provides a quantitative characterization of how a system is used [100]. Software testing based on the operational profile ensures that testing resources are focused on the most frequently used operations, and thus maximizes the reliability level that is achievable within the available testing time [100]. So, it can be a good testing strategy when safety is not an issue.

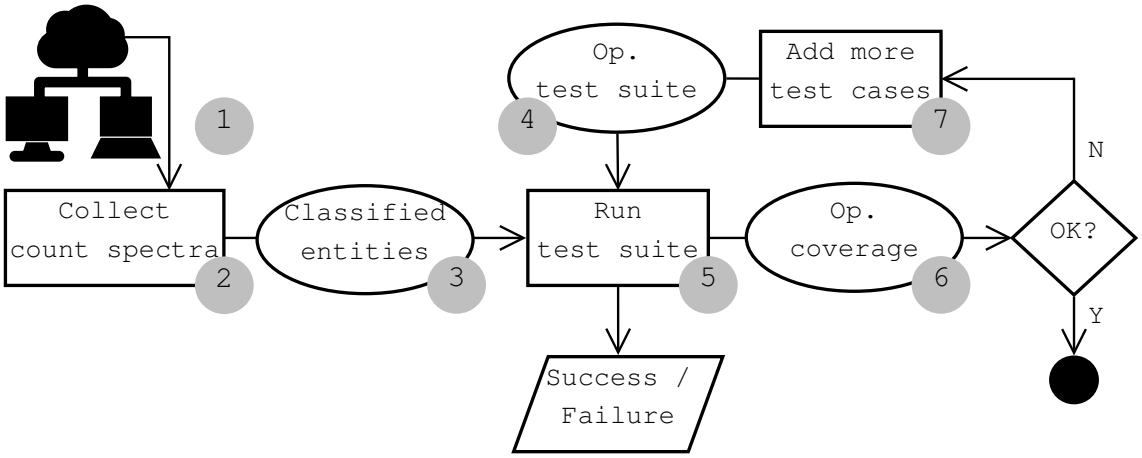


Figure 6.1: Overview of the approach

As we start from the assumption that the developer adopted an operational profile based testing strategy, we also can assume that either an operational profile is derived by domain experts during the specification stage, or that, as schematized in Figure 6.1, this profile is obtained from real world usage, e.g., by monitoring field data by means of an infrastructure such as Gamma [107].

If this developer selects a test suite from the operational profile, how can they decide whether the test suite is adequate and testing can be stopped, or otherwise more test cases should be derived? (see Figure 6.1) This is where our approach can help. It foresees two main steps:

1. Classify the entities according to their importance to the operational profile under testing

2. Calculate operational coverage based on the importance of the entities covered

We rate the importance of entities based on their frequency of usage, i.e., we make use of program spectra [57]. A program spectrum characterizes a program's behavior by recording the set of entities that are exercised as the program executes. In this work we investigated coverage of three types of entities and correspondingly adopted three types of spectra:

- **Branch-count spectrum (BCS):** for each conditional branch in a given program P, the spectrum indicates the number of times that branch was executed.

- **Statement-count spectrum (SCS):** for each statement in a given program P, the spectrum indicates the number of times that statement was executed.

- **Function-count spectrum (FCS):** for each function in a given program P, the spectrum indicates the number of times that function was executed.

Based on the spectra, we then classify the entities (step 1 of our approach) into different importance groups. In this work we used three groups: *high*, *medium*, and *low*, but other different groupings could be decided. To cluster entities into group, again, different methods could be applied. In this preliminary investigation we opted for ordering the list of entities according to their frequency and assigning the first 1/3 entities to the high importance group; the second 1/3 entities to the medium importance group; and the last 1/3 entities to the low importance group. Surely, the importance of a given entity could be assigned in many different ways and the effect of choosing one approach or another should be investigated in future work.

We calculate operational coverage (step 2) by computing the weighted arithmetic mean of the rate of covered entities according to Equation 6.1 below. Again, we observe that there exist many other different ways in which we could calculate operational coverage and Equation 6.1 is just one of the many possibilities.

$$\textbf{Operational coverage} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} w_i} \cdot 100(\%) \tag{6.1}$$

where:

n = number of importance groups
x_i = the rate of covered entities from group i
w_i = the weight assigned to group i

Notice that when a single importance group exists, or when all the existent groups have the same weight assigned, Equation 6.1 reduces to the relative coverage equation (Equation 1.2). Besides that, when the weights are normalized such that they sum up to 1, Equation 6.1 can be reduced to:

$$\sum_{i=1}^{n} w_i x_i \cdot 100(\%)$$

In this work we assigned the weights for the importance groups (the w_i of Equation 6.1) in such a way that the *medium* group is three times more important than the *low*, and the *high* group, on its turn, is three times more important than the *medium* one.

6.2 Exploratory Study

We conducted an exploratory study to assess the usefulness of adopting the proposed approach for test adequacy and selection. In this section we discuss the settings of the study. More precisely, we focus on the following research questions:

RQ1: *Does operational coverage provide a good adequacy criterion (stopping rule) for operational profile based testing?*

RQ2: *Is operational coverage useful for selecting test cases (selection criterion) for operational profile based testing?*

6.2.1 Study Subjects

For carrying out our exploratory study we used the same subjects adopted in the investigations conducted in Chapter 4 (i.e., grep, gzip, and sed). This time, the only prerequisite for choosing the subjects was that they should contain faults, either real or seeded ones, and a test suite associated with them.

We recall that each subject is available from SIR with a different number of versions and each version contains a different number of seeded faults. For this exploratory study we selected, from each subject, the version that contained the highest number of faults that could be revealed by our test pool (detailed next). We then proceeded with version 3 of grep, version 4 of gzip, and version 2 of sed. Because we adopted only one version of each subject, throughout the rest of this chapter we refer to them by the subjects' names only, without reporting the version.

Table 6.1 provides some additional details about the study subjects. Column "LoC" shows the lines of code[1] of each subject. The meaning of the last two columns is explained later on.

Table 6.1: Details about the study subjects considered in our investigations

Subject	Version	LoC	# Seeded faults	# Failing test cases	# Faults revealed
grep	v3	10124	18	1664	5
gzip	v4	5233	12	1638	5
sed	v2	9867	5	3195	4
	Total:	25224	35	6497	14

6.2.2 Operational Profile

Operational profiles can be defined in many different ways [119]. Here, following [100], we express it as the list of operations that are expected to be invoked by users along with their associated occurrence probabilities. Ideally it is developed during system specification with the participation of the system experts (e.g.: system engineers, designers, etc) and domain experts (e.g.: analysts, customers, etc). Because such an ideal operational profile was not available for the subject systems we investigated, we ourselves defined the operational profile for each subject. We accomplished this task by getting acquainted with the system (after carefully reading the user manual for the version of the subject being investigated) and by following

[1]Collected using CLOC (`http://cloc.sourceforge.net/`).

Musa's stepwise approach [100]. For experimental purposes, we stopped the process of creating the operational profile just before assigning the occurrence probabilities for each operation identified, as these are taken as an independent variable of our study.

Table 6.2 displays an excerpt of the list of operations we identified for grep. A graphical version containing the full list of operations identified (grouped according to different usage paths) is available at `http://bit.ly/op_grep`.

Table 6.2: List of operations identified for grep

ID	Description
Op_{001}	Look for a single pattern in a single input file
Op_{002}	Look for multiple patterns, obtained from a file, in a single input file
...	...
Op_{191}	Interpret the pattern as a list of fixed strings and match any of them
Op_{192}	Interpret the pattern as an extended regular expression and print the number of matching lines

6.2.3 Study Settings

Besides the study subjects obtained from SIR and the subjects' operational profiles developed by ourselves, for each of the three investigated subjects we also created a few additional artifacts required for our study. Some of them were derived once and for all:

- **Test Pool.** For each subject investigated we created a test pool containing 10k test cases uniformly distributed among the operations in the subject's operational profile.

- **Fault Matrix.** For each subject considered we run the set of 10k test cases over the baseline version first (the one without any faults enabled), and then over the faulty versions of that subject. To get a precise mapping of which test cases would reveal which faults we compile one faulty version for each seeded fault available. For grep, for example, we have one baseline version and 18 faulty versions, which accounts for 190k test cases run to generate the fault matrix. The second last column and the last column of Table 6.1 refer to the results of running the set of 10k test cases over the studied subjects: they display the number of failing test cases and of seeded faults that could be revealed, respectively.

- **Operation Matrix.** Each test case in the test pool is created for a specific operation in the operational profile and this mapping (test case, operation) is stored in the operation matrix.

Other artifacts, on the other hand, were created on a "per observation" basis. For each research question investigated, we made 500 observations for each subject. Each observation is related to a different operational profile. More precisely, for each observation we derived the following artifacts:

- **Customized Operational Profiles.** For deriving a customized operational profile we randomly select an arbitrary number of operations and randomly assign their respective occurrence probabilities in a way that the sum of the individual probabilities is equal to 1.

- **Count Spectra.**[2] For each customized operational profile we derived three different spectra: the branch-count, the statement-count, and the function-count spectrum. In order to do so, we exercise the subject with randomly generated test cases that are derived according to the operations and their respective occurrence probabilities defined in the customized operational profile. Observe that this set of randomly generated test cases is completely separated from the test pool previously introduced.

We also used the gcov[3] and lcov[4] utilities for collecting accurate coverage metrics, and our own code to automate the majority of the steps followed during this study.

6.3 Adequacy Study (RQ1)

In this section we report the results of our investigations with respect to the adoption of operational coverage as a stopping rule (RQ1). We start by describing the specific tasks and procedures associated with this exploratory study. Then, we present a summary of the study results along with an answer to our research question.

6.3.1 Tasks and Procedures

For each subject (and for each one of the 500 customized profiles created per subject) the following tasks are performed:

1. We carry out operational profile based testing by selecting the next test case to be run (from the 10k set) according to the occurrence probabilities defined in the customized operational profile.

[2]This artifact is used for the adequacy study only (Section 6.3). It does not apply for the selection study reported in Section 6.4.

[3]`https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

[4]`http://ltp.sourceforge.net/coverage/lcov.php`

2. After each test case is run, we calculate:

   (a) the traditional coverage achieved

   (b) the operational coverage achieved (calculated according to Equation 6.1)

   (c) the probability of failure for the next test case, θ

The coverage metrics (items 2a and 2b) are calculated for the three adequacy criteria considered in this study and we stop testing if none of them increase after a sequence of 10 test cases.

Calculating the probability of failure for the next test case

We illustrate the way we calculate the probability of failure (item 2c) through a simple example pointing to the artifacts described in Section 6.2.3:

- *Customized operational profile:* we assume a very simple profile including only four operations: Op_1 with occurrence probability $Pr = 0.6$, Op_2 with $Pr = 0.3$, Op_3 with $Pr = 0.1$, and Op_4 with $Pr = 0.0$.
- *Test pool:* by construction, the test pool contains the same amount of test cases for each operation in the operational profile. For this example, we assume the test pool contains 2500 test cases per operation.
- *Fault matrix:* we assume only one fault, Fault 1, and that from the Fault matrix we can see it is revealed by 100 test cases in the 10k pool.
- *Operation matrix:* from the operation matrix we can match operations to fault-revealing test cases (in the fault matrix). We assume we get: 50 failing test cases for Op_1, 30 for Op_2, 0 for Op_3, and 20 for Op_4.

Then, when no test case has been run, the probability that the next test will fail is:

$$\theta_{F_1} = \frac{(50 \times 0.6) + (30 \times 0.3) + (0 \times 0.1) + (20 \times 0)}{2500} = 0.0156$$

Because we assume that the existing faults are independent, when more faults exist, θ is the overall sum of the individual probability of failure for each fault. In the cases where coupled faults exist, however, this might not be true.

6.3.2 Study Results

A summary of the output of the tasks described in Section 6.3.1 is displayed in Table 6.3. The second column shows the span variations in the number of test cases required for performing operational profile based testing for each subject while the third column presents the average number of test cases among the 500 customized

operational profiles created for our study. As we can see, for all the subjects, testing stopped after around 60 test cases. The biggest span variation happened for gzip with some operational profiles requiring as few as 13 test cases to complete testing, whereas other profiles required 136 tests; sed had the smallest variation with the number of test cases ranging from 15 to 98.

Table 6.3: Average traditional and operational coverage achieved per subject

Subject	**#TCs span**	**Avg. #TCs**	**Branch**		**Statement**		**Function**	
			trad.	*oper.*	*trad.*	*oper.*	*trad.*	*oper.*
grep	24 to 116	64	24.7	86.3	42.4	89.2	61.5	92.5
gzip	13 to 136	56	39.9	79.3	52.0	85.4	60.3	95.5
sed	15 to 98	51	29.5	95.5	48.3	96.2	71.3	98.1
	Average:	57	31.3	87.0	47.6	90.3	64.4	95.3

Table 6.3 also displays the average traditional and operational coverage achieved grouped by different adequacy criteria. In this table, "*trad.*" and "*oper.*" stand for traditional coverage and operational coverage, respectively. Operational coverage achieved higher coverage values in all the cases. This was expected because, by construction, operational coverage targets only a subset of the entities for each operational profile, which increases the chances of providing high coverage values.

Figure 6.2 shows, for all the subjects and coverage criteria investigated, the average traditional and operational coverage achieved as the number of test cases increases. The x-axes display the number of test cases while the y-axes represent the coverage achieved. In this figure, traditional coverage and operational coverage are represented by the continuous line and the dashed line, respectively. For each test case n, its equivalent point in the curve represents the average coverage (of the 500 customized profiles) achieved after n test cases. Notice that not all the profiles finished after the same amount of test cases. For this reason it is possible to see, for gzip in particular, some fluctuation in the curve when the number of test cases gets close to the maximum number of tests required for that product.

From the graphs, the main observation is the fact that the curve of operational coverage rises sharply and achieves high coverage values even after just a few test cases. This happens due to the combination of two facts: (i) the most frequently exercised entities contribute more for the computation of the coverage achieved (because of the weight assigned to the high importance group); and (ii) the operational profile based testing strategy selects test cases that cover the most frequent entities first.

6.3.3 Answer to the Research Question

To answer RQ1 we computed, for both traditional and operational coverage, the correlation between the coverage achieved and the probability that the next test

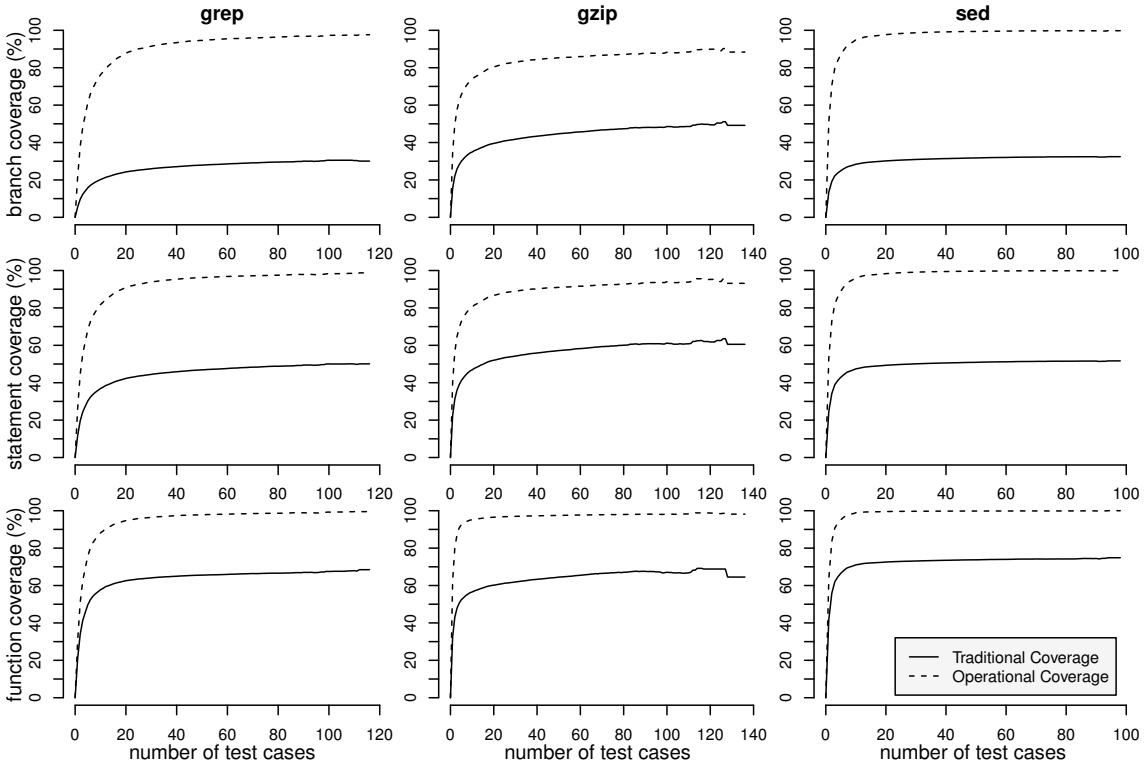


Figure 6.2: Average traditional and operational coverage achieved as the number of test cases increases

case will *not* fail ($1 - \theta$, which is the reliability of the next invocation). For doing so, we adopted the Kendall τ correlation coefficient. Kendall τ is similar to the more commonly used Pearson coefficient but it does not require the variables to be linearly related or normally distributed. By using Kendall τ we avoided introducing unnecessary assumptions about the distribution of the data.

As Kendall τ measures the similarity of the orderings of the data when ranked by each of the variables, a high correlation means that one can predict the rank of the importance of the faults revealed given the rank of the coverage achieved, which in practice is nearly as useful as predicting the absolute importance of the faults revealed.

Table 6.4 displays the Kendall τ correlation between the coverage achieved and the probability that the next test case will not fail, grouped by the different coverage criteria investigated. For interpreting the data in accordance with [64], we use the Guildford scale, in which correlations with absolute value less than 0.4 are described as “low”, 0.4 to 0.7 as “moderate”, 0.7 to 0.9 as “high”, and over 0.9 as “very high”.

As we can see, operational coverage yielded better correlation coefficients than traditional coverage for the vast majority of the cases (we highlight in bold the cases in which operational coverage performed better than traditional coverage). The only

Table 6.4: Kendall τ correlation between coverage and the probability that the next test case will not fail (all entries are statistically significant at the 99.9% level)

Subject	Branch		Statement		Function	
	trad.	*oper.*	*trad.*	*oper.*	*trad.*	*oper.*
grep	0.37	**0.40**	0.38	**0.41**	0.39	0.35
gzip	0.41	**0.45**	0.44	**0.46**	0.39	**0.44**
sed	0.39	**0.50**	0.40	**0.52**	0.35	**0.47**
Average:	0.39	**0.45**	0.41	**0.46**	0.38	**0.42**

exception was for grep when considering the function adequacy criterion, in which traditional coverage achieved a correlation coefficient of 0.39 and operational coverage produced 0.35 (statistically, they were tied as both achieved low correlation). This was the only case in which operational coverage yielded low correlation. For the remaining cases, it always achieved moderate correlation.

Traditional coverage achieved moderate correlation 3 times and in all the cases it was statistically tied with operational coverage, if we consider the correlation group; if we consider the absolute correlation coefficient achieved, though, it was always defeated by operational coverage.

6.4 Selection Study (RQ2)

After our investigations on the adoption of operational coverage for test case adequacy, we examined the usefulness of our approach for test case selection. Here we report the the specific tasks and procedures associated with the selection study as well as the results of our investigations.

6.4.1 Tasks and Procedures

For our investigations on the adoption of operational coverage for test case selection we used the 500 customized profiles per subject previously created. For each subject and for each customized profile the following tasks are performed:

1. We randomly select a subset of 1k test cases from test pool (the 10k set).

2. We calculate the faults' importance based on the customized operational profile and the test pool derived for that iteration. This calculation is done in the same way as explained in Section 6.3.1.

3. Derive one test suite using the greedy additional algorithm targeting all the entities available in the subject under testing. We refer to it as the *traditional test suite*.

4. Derive a second test suite, the *operational test suite*, again using the greedy additional algorithm, but this time targeting the most important entities for the target customized operational profile.

5. We run both test suites against the subject under testing and we measure:

   (a) the size of the derived test suites;

   (b) the traditional coverage achieved by both test suites;

   (c) the operational coverage achieved by the operational test suite;

   (d) the remaining failure probability.

Performing step 1 is important for two reasons: first, as we use a greedy algorithm to derive the test suites, if we always considered the same test pool it could be the case that the derived test suites would be very similar among different iterations. Some differences in the algorithm choices would still happen as the relevance of each entity changes on each iteration based on the customized operational profile. However, by randomly selecting a smaller set from the test pool we guarantee that the greedy algorithm will always have a different set of test cases to choose from. Second, this step changes the relative importance of each fault with respect to the test pool on every iteration. In one iteration, the 1k set may contain all the test cases that would trigger the most important fault, whereas in a different round it could contain only one test case that reveals the critical fault. This is important to observe whether or not the different approaches are able to (and to what extent) select the "best" test cases for each customized profile.

In the case that the derived test suites contain a different number of test cases, for the sake of providing a fair comparison we reduce the size of the bigger test suite down to the size of the smallest one before computing the metrics described in step 5.

6.4.2 Study Results

Figures 6.3, 6.4, and 6.5 display the boxplots of the test suites sizes derived targeting branch, statement, and function coverage, respectively. The results are grouped by the different subjects considered, and the y-axis displays the number of test cases in the devised test suites. The red dots indicate the average test suite size for each combination of subject and approach. For this metric, the lower the test suite size, the better.

Overall, the operational coverage approach defeated the traditional one in all the cases with lower median and average values. In all the cases observed, the upper quartile for the operational coverage approach is below (or very close) to the lower fence. This tells us that in about 75% of the cases the size of the test suite derived targeting operational coverage was smaller than the smallest test suite derived targeting traditional coverage.

To confirm that the observed differences in the median values were statistically significant, we performed the Wilcoxon signed-rank test. In all the cases, the null hypothesis of statistically equivalent medians was rejected. The *p-value* returned by the Wilcoxon test was smaller than 2.2e−16 for all the cases, which means that the differences in the median values are statistically significant at least at the 99.9% confidence level.

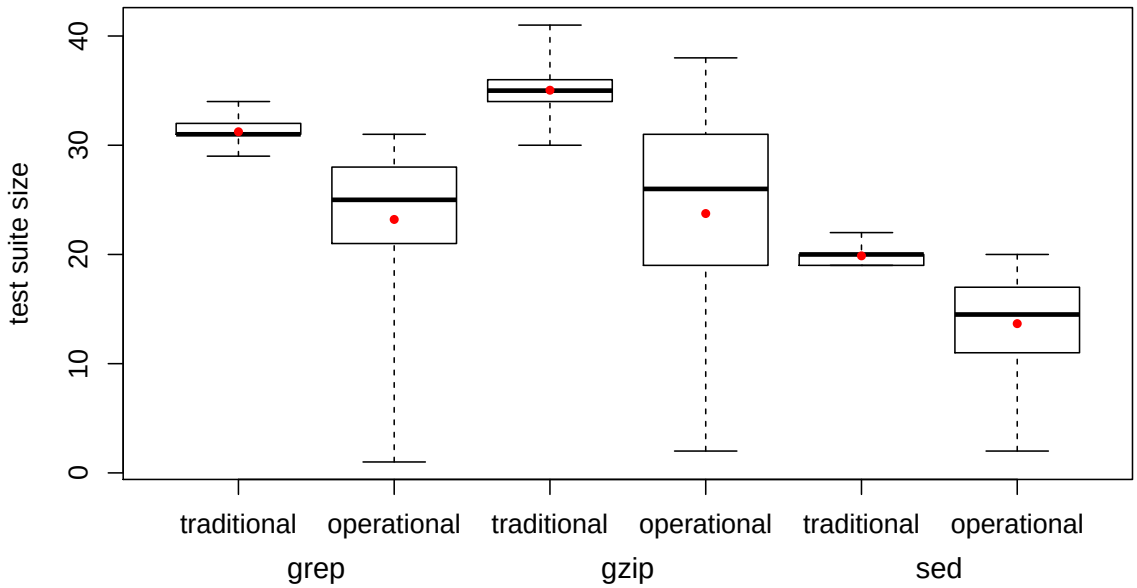


Figure 6.3: Comparison between the test suite reduction achieved by the different selection strategies when targeting Branch coverage

While targeting branch coverage (Figure 6.3) the number of test cases in the test suites derived according to the operational coverage approach varied from 1 to 31 for grep, from 2 to 38 for gzip, and from 2 to 21 for sed. For the test suites derived according to traditional coverage, the number of test cases ranged from 29 to 34 for grep, from 30 to 41 for gzip, and from 19 to 22 for sed.

With respect to the statement coverage criterion, displayed in Figure 6.4, the average number of test cases in the test suites derived for operational coverage were 22.18, 18.79, and 12.01, for grep, gzip, and sed, respectively. For the test suites derived for traditional coverage the figures were (following the same order) 28.76, 24.85, and 16.18.

For the test suites generated targeting function coverage (Figure 6.5), the average number of test cases required by operational coverage for grep was 6.28, while 6.29 test cases were required for traditional coverage. For gzip, the figures were (following the same order) 8.39 and 12.74. Finally, for sed, the average number of test cases required were 4.02 and 5.13.

Besides measuring the size of the test suites generated by the greedy algorithm when targeting different selection criteria, we also computed the traditional and

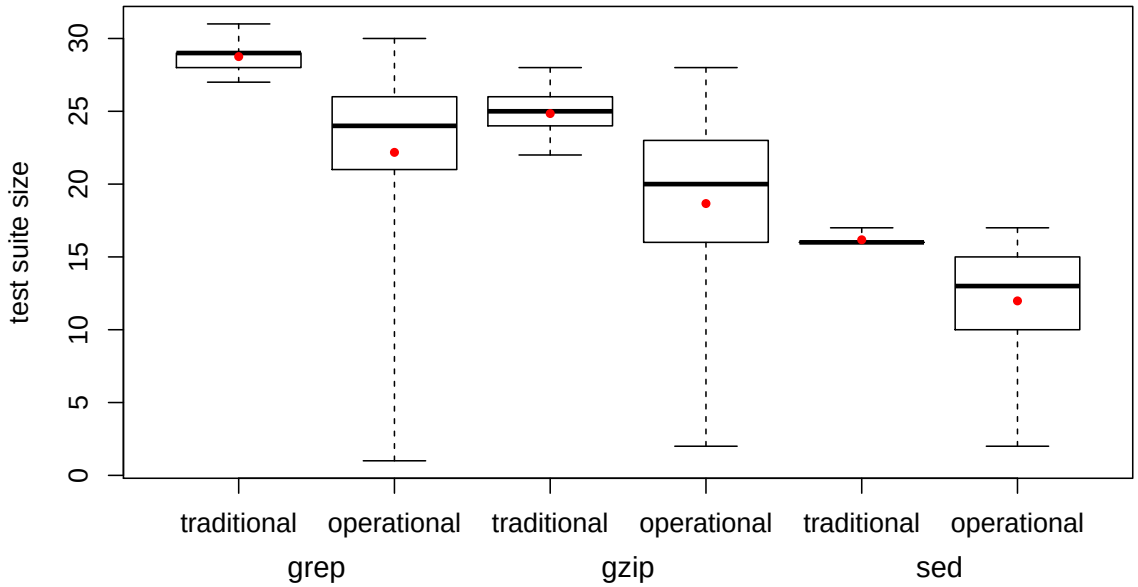


Figure 6.4: Comparison between the test suite reduction achieved by the different selection strategies when targeting Statement coverage

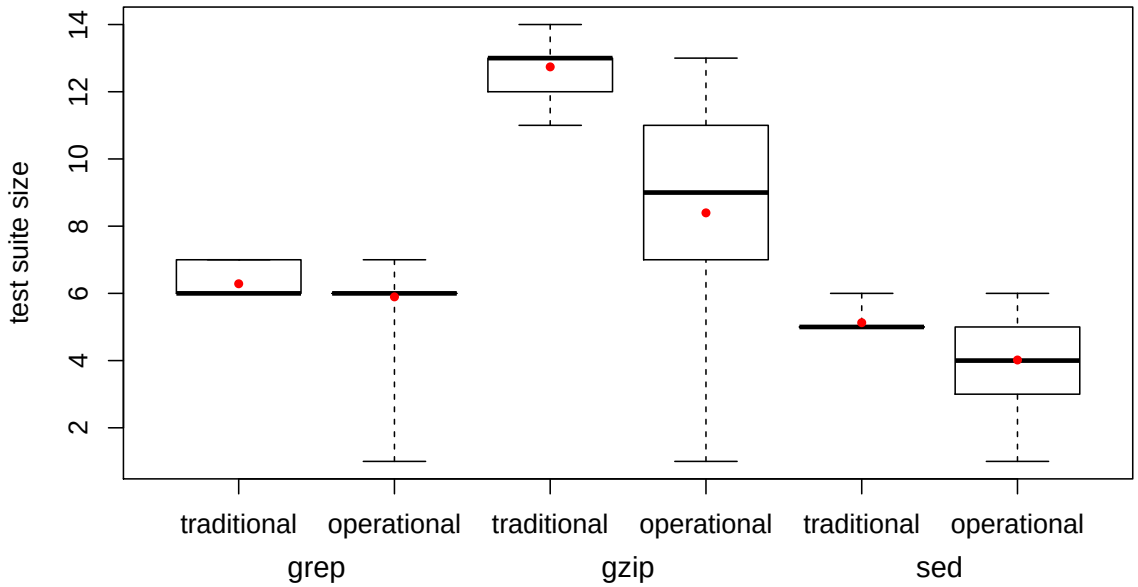


Figure 6.5: Comparison between the test suite reduction achieved by the different selection strategies when targeting Function coverage

operational coverage achieved by them. The results are available in Table 6.5 and they are grouped by the different subjects and coverage criteria considered in this study. In this table, column "*trad.*" displays the traditional coverage achieved by the traditional test suite, while column "*oper.*" shows the operational coverage achieved by the operational test suite. As we were also interested in understanding

how the test suite derived targeting the most important entities only would perform in traditional terms, we also computed the traditional coverage achieved by the operational test suite (this information is available in the column "*trad**").

Table 6.5: Average coverage (in %) achieved by the test suites derived according to different selection criteria

Subject	Branch			Statement			Function		
	trad.	*oper.*	*trad**	*trad.*	*oper.*	*trad**	*trad.*	*oper.*	*trad**
grep	32.43	99.69	30.86	51.79	99.82	50.43	68.33	99.71	66.47
gzip	55.59	99.82	50.40	69.04	99.87	63.82	69.26	99.87	66.23
sed	32.46	99.99	31.69	51.61	99.99	50.79	74.12	100.00	73.35
Average:	40.16	99.84	37.65	57.48	99.90	55.01	70.57	99.86	68.68

As we can see in Table 6.5, the average operational coverage was very high in all the cases observed. As operational coverage targets only a subset of the entities from the program under testing, and considering the relatively big pool of test cases to select from (the 1k set), it was already expected that the operational test suites would achieve 100% operational coverage in many cases. Indeed, when targeting function coverage for sed, the operational test suites achieved 100% coverage of the in-scope entities in each one of the 500 observations.

When the coverage of the operational test suites was measured in the traditional way, column "*trad**", the average coverage achieved was always lower (but very similar) to the one obtained by the traditional test suites.

6.4.3 Answer to the Research Question

To answer RQ2, we computed, for the traditional and operational test suites, the remaining failure probability, i.e., the probability that the next test case would fail after finishing running the test suites. Reaching a 0 for this metric means that all the relevant faults for the customized operational profile have been revealed by the test suite. Thus, the lower the value, the better.

Table 6.6 displays the remaining failure probability grouped by subject and coverage criteria. The results achieved by the traditional and operational test suites are shown in columns "*trad.*" and "*oper.*", respectively. We highlight in bold the cases in which operational coverage outperformed traditional coverage.

Operational coverage was defeated by traditional coverage for grep when targeting function coverage, and for gzip when directing test selection for branch coverage. For all the other cases operational coverage performed better than the traditional one. When considering the overall average, operational coverage exceeded traditional coverage for the three coverage criteria investigated in this study.

Despite the fact that operational coverage performed better in the majority of the cases when considering the average remaining failure probability, when we look at the

Table 6.6: Remaining failure probability (in %) after test suite execution (all entries are statistically significant at the 95% confidence level)

Subject	Branch		Statement		Function	
	trad.	*oper.*	*trad.*	*oper.*	*trad.*	*oper.*
grep	2.720	**0.907**	2.180	**0.804**	7.113	7.815
gzip	0.003	0.063	0.056	**0.043**	1.200	**0.966**
sed	0.205	**0.147**	0.306	**0.174**	15.125	**13.682**
Average:	0.976	**0.372**	0.847	**0.340**	7.813	**7.488**

individual results for each customized operational profile, operational and traditional approaches were tied in more than half of the cases. This was an unforeseen result as we were expecting that the test suites derived targeting a specific operational profile would had performed much better than the ones generated without further guidance regarding the relevance of the entities of the program under testing.

One possible explanation for that unexpected result is *collateral coverage*. It may have influenced the results in two ways: (i) for operational coverage, by mapping white-box entities to existing test cases, it is possible that the greedy algorithm selects *not relevant test cases* (a test case that covers some of the target entities but that is not related to the target operation). Traditional coverage (ii), on the other hand, may have "benefited" from this collateral coverage effect as the selection of a test case targeting a particular set of entities will most likely also satisfy the coverage of further entities by accident. Thus, even if the most important entities for a given operational profile are not the real target of the greedy algorithm aiming traditional coverage, many of them may be covered unintentionally by the first test cases selected.

6.5 Discussion

The exploratory studies conducted in this chapter showed that:

- As an *adequacy criterion*, operational coverage is statistically better than traditional coverage for operational profile based testing in the majority of the cases.

- As a *selection criterion* for operational profile based testing, operational coverage outperformed the traditional one on average, but both approaches were tied in the majority of the cases.

In the following we further discuss potential costs and benefits of the approach.

On the costs of the approach

The cost of classifying the entities according to their importance (the first step of our approach) will depend on how the operational profile is derived. For the case advised in Figure 6.1 in which the operational profile can be derived by monitoring field data, the count spectrum (bullet 3, Figure 6.1) required for defining the relevance of the entities might even be readily available, or otherwise can be obtained by using mining techniques to capture the frequencies of the entities being exercised by the users.

Regarding the cost of computing the operational coverage (bullet 6, Figure 6.1), as for any coverage criterion, operational coverage presupposes that the code is instrumented so to allow the identification of the entities exercised. So the cost of applying the operational coverage equation is comparable to any other coverage metric.

Operational coverage vs. traditional coverage

The proposed operational coverage may be particularly helpful for test suite augmentation. After the developer has derived the test suite to verify a given program, two main things could happen: (i) for all the code exercised by the program's users there exist at least one test case in the test suite that covers that code; or (ii) there exist some code exercised by the program's users for which there does not exist any test case in the test suite.

Operational coverage would acknowledge the former case by yielding 100% coverage. For the latter case, it would provide an assessment of the extent to which the derived test suite is adequate to that specific operational profile. Moreover, it would also provide the precise portions of code that are *actually exercised* by the program's users and that are not covered by the test suite, guiding the test suite enhancement process towards the real usage of the program.

Regarding traditional coverage, for the case (i), if the test suite does not achieve 100% traditional coverage — probably the case — the developer cannot tell whether or not all the code that is relevant to the program's users has been covered. Similarly, for case (ii) traditional coverage does not help the developer unless they are willing to augment the test suite until 100% traditional coverage is reached, which may be impractical even for small programs.

Even though we believe that operational coverage may be a promising approach for operational profile based test case selection, its adoption may not be as straightforward as it was for adequacy criterion. As anticipated in Section 6.4.3, one of the issues involved is that of collateral coverage. To overcome this effect, some fairly sophisticated selection approach that takes into account not only the relevance of the entities to be covered, but also the way the candidate test cases cover those entities, needs to be devised. We plan to investigate this as part of our future work.

6.6 Threats to Validity

The exploratory study results should be interpreted by keeping in mind the following potential threats to validity:

Subject representativeness: In this work we investigated the proposed operational coverage on three C programs from the SIR repository. Additional studies using a range of diversified subjects should be conducted for better representativeness.

Subjects' operational profile: Because the subjects' operational profiles were not readily available, we had to develop them ourselves. It is possible, then, that the set of operations we identified is not a complete list of operations that could be performed by real users. We controlled this threat by carefully reading the subjects' documentation to understand well how they could be used before developing the operational profiles.

Customized profiles representativeness: We derived the customized operational profiles by randomly selecting the target operations and their respective occurrence probabilities, which may have results into unrealistic profiles. To control this threat, we created 500 customized operational profiles for each subject investigated expecting that the effect of possible unrealistic profiles would be minimized with a big number of observations. Moreover, since both metrics (traditional and operational coverage) are captured for each customized profile, we do not see how such threat could produce different impacts on our evaluations in a systematic way, and thus influence the results biased to the benefit of one approach or another.

Faults representativeness: The subjects considered in our study contained seeded faults and subjects with real faults might yield different results. Control for this threat can be achieved only by conducting additional studies using subjects with real faults.

Operational coverage calculation: The operational coverage is influenced by parameters that allow a high level of customization: the importance assigned to the entities, the weight assigned to the importance groups, and the equation itself. Control for this threat can be achieved only by conducting additional studies using different configurations.

6.7 Related Work

In this chapter we introduced a coverage criterion for operational profile based testing. The work reported here is mainly related coverage testing, software reliability and operational testing. A preliminary discussion for these topics was provided in

Chapter 2 (Sections 2.1 and 2.4). Besides that, the introduced coverage criterion, the Operational Coverage, is also related to the notion of relative coverage already presented in Chapter 4. Here, we follow the same approach adopted in the previous chapter and we present only the related work that has not been covered in any of the previous chapters. The additional related work for this chapter include research from the following topics: operational testing, program spectra, sensitivity analysis, and continuous testing.

Operational Testing. Our work is related to operational testing, as pursued in, e.g., Musa's SRET (Software Reliability-Engineered Testing) approach [101]. In SRET test cases are selected from the user's operational profile, thus those inputs that are forecast to be more often invoked in use are also more stressed in testing. By classifying the entities according to their importance to the operational profile under testing and by calculating coverage based on the importance of the entities covered, we also aim at targeting the input subdomain that is the most relevant for the user, while giving lower priority to inputs that are not or seldom exercised. SRET uses a statistical approach for selecting test cases based on the user's operational profile. In our approach, we assume that such an operational profile would be available. In the case of its absence, we propose that field data could be exploited in the form of count spectra to capture the frequency at which entities are exercised by real users.

Program Spectra. Program spectra [57] have been used extensively in software analysis. Beyond the original application in program optimization [11], more recently code profiling information has been used to analyze the executions of different versions of code, e.g. in regression testing [136], and to compare traces of failed and successful runs in fault diagnosis [1]. Here we propose to exploit traces information to tune coverage measures onto the usage profile. This is a novel application of spectra in operational profile based testing that has never been tried, and could be exploited in many ways.

Sensitivity Analysis. There are similarities between operational coverage and the former notion of sensitivity analysis by Voas and coauthors [127]. Sensitivity was defined as the probability that a particular program "location" could reveal a possible fault under a specified input profile, where a location is a unit of code that can change a variable's value (it may roughly correspond to a statement, although some statements could also contain more locations). We do not consider specifically program locations, however similarly to sensitivity analysis, we profile program entities under the operational usage profile to understand their impact on probability of failure. So, motivations are similar, even though we then propose a different application of the notion.

Continuous Testing. Modern pervasive and interconnected networks and huge advances in potential to collect and analyze big data make it thinkable that developers continue testing and maintenance of deployed software by exploiting usage profiling information [104]. This requires the establishment of proper infrastructures [107], but can provide many opportunities for improved testing and analysis techniques [31]. For example, in [105] field data are exploited for impact analysis and regression testing; in [70] field failures data are used to support in-house debugging and fault localization. More in general, costs and benefits of profiling deployed software are empirically studied in [31], and the results confirmed their large potential. Hence, our approach is one among the many opportunities provided by field data to improve testing techniques.

6.8 Concluding Remarks

In this chapter we have introduced operational coverage, which measures code coverage taking into account whether and how the entities are relevant with respect to a user's operational profile. We propose that the entities to be covered (even basic ones, such as function, statement or branch) should be weighted based on their relevance according to a usage profile.

In the previous instantiations of relative coverage presented in preceding chapters (i.e., relevant coverage and social coverage), we distinguished between in-scope and out-of-scope entities by assigning a weight of 1 or 0 to the entities. In this chapter, instead, we considered frequency of coverage and accordingly propose to weight entities using a count spectrum. Operational coverage is the very first attempt to tune coverage testing based on program count spectra. In particular, here we used a simple approach to map count spectrum of branches, statements, and functions into a coverage measure.

Many studies evaluating the effectiveness of coverage measures [120, 64, 50, 75] consider that all the faults (or mutants) have the same importance, and do not take into account their respective probability of failure. Considering the possible impact of faults detected, as we do here, provides a more meaningful picture when the software under test is going to be used under different profiles, and thus the effectiveness of coverage measures for a same software may vary depending on who is the future user.

Our exploratory study showed that operational coverage is better correlated than traditional coverage with the probability that the next test case derived according to the user's profile will not fail. This result suggests that our approach could provide a good stopping rule for operational profile based testing. With respect to test case selection, our investigations revealed that operational coverage outperformed the traditional one in terms of test suite size and fault detection capability when we look at the average results. However, as anticipated in Section 6.5, more empirical

studies are required to better understand the potential usage of operational coverage for test case selection.

7

Conclusions

In this thesis we have introduced a new way of approaching coverage testing with the objective of improving the cost-effectiveness of the testing process. We propose to do this by focusing the testing activities on the program parts that are relevant according to the testing scope.

Since its conception, coverage testing has been approached in a very rigid nature based on the idea that one cannot consider a program to be adequately tested if certain entities have never been executed by any test data. As we demonstrated in this thesis, however, this inflexible definition makes traditional coverage approaches useless for some testing contexts. This happens because these approaches cannot help developers to neither (i) decide when to stop testing nor (ii) to augment their test suites when the measured coverage is not deemed sufficient.

In their seminal work "An Applicable Family of Data Flow Testing Criteria" [37], Frankl and Weyuker proposed to circumvent the problem of non-applicability of the data flow testing criteria by requiring the test data to exercise *only* those definition-use associations that are executable. In other words, they proposed, as simple as it may sound, to change the denominator of the coverage equation to exclude the infeasible entities. This little change had a big impact on the software testing research: achieving 100% coverage was now possible and this opened the way for the proposal, evaluation and comparison of a enormous number of different adequacy criteria. Our work presented in this thesis follows the same direction and proposes another change in the denominator of the coverage equation. This time, we propose the exclusion of entities that are *not relevant* to a given testing context, even if they are perfectly feasible.

By proposing to focus the testing efforts on the relevant entities we do not advocate that less testing should be carried out. In the same way that any adequacy criterion existing in the literature represents the minimal standards for testing a program, our proposal is not different and, if the available resources allow, of course more testing should be performed to gain confidence about the program being tested. If the testing resources are constrained, on the other hand (e.g., due to a tight schedule), by directing the efforts to the relevant entities, our approach ideally ensures that the available resources are well spent on the most important parts of the program under testing for a given testing context.

In comparison with traditional approaches for calculating coverage, relative coverage requires only one extra step to be performed, i.e., the identification of the *in-scope entities*. Before testing a given program, one should be aware of the general testing goals and the context in which the program is going to be used, that is, the *testing scope*. For this reason, as stated in the introduction, defining the testing scope was not a research objective of this thesis. Rather, we assumed that the information regarding the specific testing context would be available, either informally in the developer's minds or in some formal specification document. The real challenge is, then, to map the *testing scope* into *in-scope entities*. This step could be achieved by asking the testers themselves to explicitly specify the entities of interest as proposed in [13, 33]. The problem with this approach, however, is that this information may not be easily a priori known by the testers. In this thesis we proposed to automate this step in different ways depending on the artifacts (e.g., source code) availability. In Chapter 4 (source code available), we adopted Dynamic Symbolic Execution (DSE) to analyze the code dynamically with its exploration being guided by the possible constraints over the input domain. Next, in Chapter 5 (source code *not* available), the set of in-scope entities was defined based on the coverage information collected from similar users. Finally, in Chapter 6 (source code and field data available), we proposed the use of the count spectra obtained from real world usage for the identification of the in-scope entities. Having identified the set of in-scope entities, we can use that information for calculating relative coverage as we would do with traditional approaches.

The examples provided in the previous chapters demonstrating the applicability of relative coverage are not exhaustive. Rather, they are just a small sample that aims at providing a glimpse of what could be achieved by adopting our approach. Due to the pace of development and the breadth of research in software testing, illustrating the adoption of relative coverage for the many different coverage criteria, testing contexts, programming languages, etc., would probably be impossible, and certainly beyond the scope of this thesis. Still, through the small sample investigated in this thesis we have been able to explore the usage of relative coverage along with commonly used testing techniques.

More precisely, in Chapter 4 we investigated the usefulness of relative coverage for test case prioritization, selection and minimization in the context of software reuse. In that chapter, we demonstrated how our approach could be integrated with different greedy-, similarity-, and search-based techniques. Our empirical evaluation showed that in test suite prioritization we could improve the average rate of faults detected when considering faults that were in scope, while remaining competitive when considering all faults; in test case selection and minimization we could considerably reduce the test suite size, with small to no extra impact on fault detection effectiveness considering both in-scope and all faults.

In Chapter 5, we proposed an instantiation of the relative coverage idea for the context of service-oriented architecture. In that chapter, we showed how the relative coverage approach could be implemented even when the source code is not available.

We evaluated the proposed approach in the context of a real world system in terms of its ability to accurately predict the operations that would be relevant to a given user (the in-scope entities) based on the analysis of the operations invoked by similar users. Our exploratory study showed that, in similar environments, social coverage could provide a better support to testers than traditional coverage.

Finally, in Chapter 6 we suggested that relative coverage could also be adopted for test adequacy and selection criteria for operational profile based testing. In comparison with the other instantiations of relative coverage, in that chapter we went one step further and we proposed an approach that not only classified the entities as in-scope or out-of-scope, but also assigned weights to the in-scope entities according to their expected usage in the system under testing. Our exploratory study showed that operational coverage was better correlated than traditional coverage with the probability that the next test case derived according to the user's profile would not fail, suggesting that our approach could provide a good stopping rule for operational profile based testing. With respect to test case selection, the results were not conclusive and more empirical studies are required to better understand the potential usage of operational coverage for test selection.

While conducting our research we have identified many opportunities of future work. In the short term, we would like to investigate different approaches for the identification of the in-scope entities when the source code is available. In Chapter 4, we adopted DSE motivated by the fact that this technique is very actively investigated and several tools are available. However, different approaches such as program slicing, or the application of a reachability algorithm on the static call graph, for example, could also be explored. Besides that, in our studies we considered only dynamic testing, which involves exercising a program with one or more input vectors (test cases) and assessing the outputs produced. Other validation techniques, which do not involve the execution of test cases, such as static analysis and formal verification were out of the scope of this thesis, but our notions of testing scope and relative coverage could also be explored in future work. With the exception of the study about "Social Coverage", reported in Chapter 5, we mainly explored coverage criteria that are entirely white-box or program-based (i.e., independent from the specification). Thus, as an additional future work direction we would like to further investigate the usefulness of our approach for black-box coverage criteria. Another interesting research direction is to investigate the use of the in-scope entities for test case generation. With the DSE tool we selected for carrying out the experiments reported in Chapter 4, test case generation was given for free as test cases would be generated during the symbolic exploration. However, if we adopt a different approach for identifying the set of in-scope entities, *how effective would be a test suite generated targeting the set of in-scope entities?* This is an interesting question that we would like to answer as part of our future work. Finally, we would like to further explore the potential impact of our work to the oracle problem. In Section 2.1 we anticipated that, by focusing the testing efforts on the entities that are relevant to a particular testing context, it is expected that the number of required oracles would

be reduced, which would imply in reducing the effort required for deriving them. However, there may be more profound ways in which our work relates to the oracle problem and we would like to investigate them in the future.

The proposed approach creates many research and development opportunities for both academia and industry. From the academic point of view, most of the research done related to coverage testing would be applicable for relative coverage as well (e.g., the definition of new criteria, test case generation, test case selection). From the industrial perspective, one of the main opportunities is the development of tools for providing relative coverage metrics. As it is well known, collecting coverage metrics incurs instrumentation costs and a lot of research has been devoted to the development of efficient instrumentation methods. We believe that our approach could help to reduce costs by instrumenting only the parts of the code that are of interest for a given testing scope.

Notwithstanding in four decades of software testing literature hundreds of proposals can be found of new coverage criteria aimed at improving fault-finding effectiveness, the notion of relative coverage presented here provides a completely new point of view. By adopting relative coverage we move from a rigid concept to a flexible notion that customizes the coverage information according to the testing context. Such a novel perspective paves the way to many new avenues for improving the cost-effectiveness of testing, yet all to be explored.

Bibliography

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[2] Edward N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, Jan 1984.

[3] S. Akimoto, R. Yaegashi, and T. Takagi. Test case selection technique for regression testing using differential control flow graphs. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*, pages 1–3, June 2015.

[4] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742–762, 2010.

[5] Everton L. G. Alves, PatrÃŋcia D. L. Machado, Tiago Massoni, and Miryung Kim. Prioritizing test cases for early detection of refactoring faults. *Software Testing, Verification and Reliability*, pages n/a–n/a, 2016.

[6] Pedro Andrade, Leonardo Candela, Donatella Castelli, Andrea Manzi, and Pasquale Pagano. The D4Science Production Infrastructure. Technical Report 2009-TR-054, Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR, 2009.

[7] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[8] IEEE Standards Association et al. Standard glossary of software engineering terminology. *lEEE Std*, pages 610–12, 1990.

[9] Xiaoying Bai, Yongbo Wang, Guilan Dai, Wei-Tek Tsai, and Yinong Chen. A framework for contract-based collaborative verification and validation of web services. In *Proc. of the 10th International Conference on Component-based Software Engineering*, CBSE'07, pages 258–273, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, TAV3, pages 210–218, New York, NY, USA, 1989. ACM.

[11] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: The showdown. In *Proc. 25th Symp. on Principles of Prog. Languages*, POPL'98, pages 134–148, New York, NY, USA, 1998. ACM.

[12] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5):507–525, 2015.

[13] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Whitening SOA testing. In *Proc. of the 7th joint European software engineering conf. and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 161–170, New York, NY, USA, 2009. ACM.

[14] Cesare Bartolini, Antonia Bertolino, Sebastian Elbaum, and Eda Marchetti. Bringing white-box testing to service oriented architectures through a service oriented approach. *Journal of Systems and Software*, 84(4):655–668, 2011.

[15] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. Softw. Eng.*, 13(12):1278–1296, December 1987.

[16] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[17] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, pages 65–88. Springer, 2008.

[18] David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[19] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. Creating vulnerability signatures using weakest preconditions. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pages 311–325. IEEE, 2007.

[20] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[21] Cristian Cadar, Vijay Ganesh, Peter Pawlowski, David Dill, and Dawson Engler. Exe: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the ACM Conference on Computer and Communications Security*. Citeseer, 2006.

[22] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, February 2013.

[23] Cagatay Catal and Deepti Mishra. Test case prioritization: A systematic mapping study. *Software Quality Control*, 21(3):445–478, September 2013.

[24] Ting Chen, Xiao song Zhang, Shi ze Guo, Hong yuan Li, and Yue Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Gen. Comp. Systems*, 29(7):1758 – 1773, 2013.

[25] TY Chen and MF Lau. Heuristics towards the optimization of the size of a test suite. In *Proceedings of the 3rd international conference on software quality management*, volume 2, pages 415–424, 1995.

[26] Paolo Cremonesi, Franca Garzotto, and Roberto Turrin. User effort vs. accuracy in rating-based elicitation. In *Proceedings of the Sixth ACM Conference on Recommender Systems*, RecSys '12, pages 27–34, New York, NY, USA, 2012. ACM.

[27] F. Del Frate, P. Garg, A.P. Mathur, and A. Pasquini. On the correlation between code coverage and software reliability. In *Proc. 6th Int. Symposium on Sw Reliability Engineering*, pages 124–132, Oct 1995.

[28] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[29] Hyunsook Do and Gregg Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 141–151, New York, NY, USA, 2006. ACM.

[30] Joe W. Duran and S.C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438–444, 1984.

[31] Sebastian Elbaum and Madeline Diep. Profiling deployed software: Assessing strategies and testing opportunities. *IEEE Trans. Softw. Eng.*, 31(4):312–327, April 2005.

[32] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[33] M.M. Eler, Antonia Bertolino, and P.C. Masiero. More testable service compositions by test metadata. In *Service Oriented System Engineering (SOSE), 2011 IEEE 6th International Symposium on*, pages 204–213, 2011.

[34] Emelie Engström and Per Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53(1):2–13, January 2011.

[35] W.B. Frakes and S. Isoda. Success factors of systematic reuse. *Software, IEEE*, 11(5):14–19, Sept 1994.

[36] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[37] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, October 1988.

[38] P.G. Frankl, R.G. Hamlet, Bev Littlewood, and L. Strigini. Evaluating testing methods by delivered reliability. *IEEE Trans. Softw. Eng.*, 24(8):586–601, Aug 1998.

[39] P.G. Frankl and S.N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *Software Engineering, IEEE Transactions on*, 19(8):774–787, 1993.

[40] Phyllis G Frankl and Stewart N Weiss. An experimental comparison of the effectiveness of the all-uses and all-edges adequacy criteria. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 154–164. ACM, 1991.

[41] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In MatthewB. Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Sw Eng.*, volume 4422 of *LNCS*, pages 291–305. Springer, 2007.

[42] JerryZeyu Gao and Ye Wu. Testing component-based software – issues, challenges, and solutions. In Rick Kazman and Daniel Port, editors, *COTS-Based Software Systems*, volume 2959 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin Heidelberg, 2004.

[43] gCube Development Team. gCube Website. `https://www.gcube-system.org`, 2008.

[44] A. Gladston, H. Khanna Nehemiah, P. Narayanasamy, and A. Kannan. Test suite reduction using hgs based heuristic approach. *Computing and Informatics*, 34(5):1113–1132, 2015. cited By 0.

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[46] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[47] PI Good. Robustness of Pearson correlation. *Interstat*, 15(5):1–6, 2009.

[48] J.B. Goodenough and S.L. Gerhart. Toward a theory of test data selection. *IEEE Trans. on Software Engineering*, SE-1(2):156–173, 1975.

[49] J.B. Goodenough and S.L. Gerhart. Toward a theory of testing: Data selection criteria. *Current Trends in Programming Methodology*, 2(2):44–79, 1977.

[50] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proc. 36th Int. Conference on Software Engineering*, ICSE 2014, pages 72–82, New York, NY, USA, 2014. ACM.

[51] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence (program testing). *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, December 1990.

[52] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Trans. Softw. Eng. Methodol.*, 24(2):10:1–10:31, December 2014.

[53] M. Harman. Making the case for MORTO: Multi objective regression test optimization. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 111–114, March 2011.

[54] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 182–191, April 2010.

[55] Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, and Joachim Wegener. The impact of input domain reduction on search-based test data generation. In *Proceedings of the the 6th joint meeting of the European software*

engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 155–164. ACM, 2007.

[56] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[57] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Proc. PASTE '98*, pages 83–90, 1998.

[58] Hadi Hemmati and Lionel Briand. An industrial investigation of similarity measures for model-based test case selection. In *Proc. of the 2010 IEEE 21st Int. Symposium on Software Reliability Engineering*, ISSRE '10, pages 141–150, Washington, DC, USA, 2010. IEEE Computer Society.

[59] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. Comparing white-box and black-box test prioritization. In *Proc. of the 38th International Conference on Software Engineering*, ICSE 2016, 2016. "to appear".

[60] Charitha Hettiarachchi, Hyunsook Do, and Byoungju Choi. Risk-based test case prioritization using a fuzzy expert system. *Information and Software Technology*, 69:1 – 15, 2016.

[61] Bill Hetzel. *The complete guide to software testing*. QED Information Sciences, Inc., Wellesley, MA, USA, 2nd edition, 1988.

[62] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Softw. Eng.*, 2(3):208–215, May 1976.

[63] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EUA, 2004.

[64] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, 2014. ACM.

[65] Muhammad Jaffar-ur Rehman, Fakhra Jabeen, Antonia Bertolino, and Andrea Polini. Testing software components for integration: A survey of issues and techniques: Research articles. *Softw. Test. Verif. Reliab.*, 17(2):95–133, June 2007.

[66] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 247–258, New York, NY, USA, 2016. ACM.

[67] A.K. Jena, S.K. Swain, and D.P. Mohapatra. Model-based test-suite minimization using modified condition/decision coverage (mc/dc). *International Journal of Software Engineering and its Applications*, 9(5):61–74, 2015. cited By 0.

[68] Yue Jia and Mark Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, 29-31 August 2008.

[69] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and TH Tse. Adaptive random test case prioritization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 233–244. IEEE, 2009.

[70] Wei Jin and Alessandro Orso. F3: fault localization for field failures. In *Proc. ISSTA*, pages 213–223. ACM, 2013.

[71] David S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 38–49, New York, NY, USA, 1973. ACM.

[72] J.A. Jones and M.J. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proc. IEEE Int. Conf. on Software Maintenance*, pages 92–101, 2001.

[73] Jussi Kasurinen, Ossi Taipale, and Kari Smolander. Test case selection and prioritization: Risk-based or design-based? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10. ACM, 2010.

[74] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[75] P.S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *Proc. SANER*, pages 560–564, March 2015.

[76] Xuan Nhat Lam, Thuc Vu, Trong Duc Le, and Anh Duc Duong. Addressing cold-start problem in recommendation systems. In *Proceedings of the 2Nd International Conference on Ubiquitous Information Management and Communication*, ICUIMC '08, pages 208–211, New York, NY, USA, 2008. ACM.

[77] Rikard Land, Daniel Sundmark, Frank Lüders, Iva Krasteva, and Adnan Causevic. Reuse with software components - a survey of industrial state of practice. In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, ICSR '09, pages 150–159, Berlin, Heidelberg, 2009. Springer-Verlag.

[78] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[79] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *Software Engineering, IEEE Transactions on*, 33(4):225–237, 2007.

[80] Chu-Ti Lin, Kai-Wei Tang, and Gregory M. Kapfhammer. Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests. *Information and Software Technology*, 56(10):1322 – 1344, 2014.

[81] Jacques-Louis Lions. Ariane 5 flight 501 failure, 1996.

[82] Michael R Lyu et al. *Handbook of software reliability engineering*. IEEE computer society press CA, 1996.

[83] Prateeva Mahali, Arup Abhinna Acharya, and Durga Prasad Mohapatra. *Computational Intelligence in Data Mining—Volume 2: Proceedings of the International Conference on CIDM, 5-6 December 2015*, chapter Test Case Prioritization Using Association Rule Mining and Business Criticality Test Value, pages 335–345. Springer India, New Delhi, 2016.

[84] Brian Marick. How to misuse code coverage. In *Proc. 16th Int. Conf. on Testing Comp. Sw.*, pages 16–18, 1999.

[85] M. D. McIlroy. Mass-produced software components. In *Software Engineering: A Report on a Conf. Sponsored by the NATO Science Committee*, pages 138–155. NATO, 1969. `http://www.cs.dartmouth.edu/~doug/components.txt`.

[86] Phil McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004.

[87] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 13–24, New York, NY, USA, 2006. ACM.

[88] Phil McMinn, Muzammil Shahbaz, and Mark Stevenson. Search-based test input generation for string data types using the results of web queries. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 141–150. IEEE, 2012.

[89] Breno Miranda. A proposal for revisiting coverage testing metrics. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 899–902, 2014.

[90] Breno Miranda and Antonia Bertolino. Social coverage for customized test adequacy and selection criteria. In *9th International Workshop on Automation of Software Test, AST 2014*, pages 22–28, 2014.

[91] Breno Miranda and Antonia Bertolino. Improving test coverage measurement for reused software. In *41st Euromicro Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2015, Madeira, Portugal, August 26-28, 2015*, pages 27–34, 2015.

[92] Breno Miranda and Antonia Bertolino. Does code coverage provide a good stopping rule for operational profile based testing? In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, pages 22–28, New York, NY, USA, 2016. ACM.

[93] Breno Miranda and Antonia Bertolino. Scope-aided test prioritization, selection and minimization for software reuse. *Journal of Systems and Software*, 2016.

[94] Parastoo Mohagheghi, Reidar Conradi, Ole M. Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[95] D. Mondal, H. Hemmati, and S. Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10, April 2015.

[96] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *IEEE Trans. Softw. Eng.*, 28(4):340–357, April 2002.

[97] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[98] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, April 1998.

[99] J. D. Musa. Introduction to software reliability engineering and testing. In *Software Reliability Engineering - Case Studies, 1997. Proceedings., The Eighth International Symposium on*, pages 3–12, Nov 1997.

[100] John D Musa. Operational profiles in software-reliability engineering. *Software, IEEE*, 10(2):14–32, 1993.

[101] John D. Musa. Software-reliability-engineered testing. *IEEE Computer*, 29(11):61–68, 1996.

[102] Glenford J. Myers. *The art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[103] Simeon C. Ntafos. An evaluation of required element testing strategies. In *Proceedings of the 7th international conference on Software engineering*, ICSE '84, pages 250–256, Piscataway, NJ, USA, 1984. IEEE Press.

[104] Alessandro Orso. Monitoring, analysis, and testing of deployed software. In *Proc. FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 263–268, New York, NY, USA, 2010. ACM.

[105] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. ESEC/FSE-11*, pages 128–137. ACM, 2003.

[106] Alessandro Orso, Mary Jean, and David Rosenblum. Component metadata for software engineering tasks. In Wolfgang Emmerich and Stefan Tai, editors, *Engineering Distributed Objects*, volume 1999 of *Lecture Notes in Computer Science*, pages 129–144. Springer Berlin Heidelberg, 2001.

[107] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *Proc. Int. Symp. on Sw. Testing and Analysis*, ISSTA '02, pages 65–69. ACM, 2002.

[108] A. Panichella, R. Oliveto, M. D. Penta, and A. De Lucia. Improving multiobjective test case selection by injecting diversity in genetic algorithms. *IEEE Transactions on Software Engineering*, 41(4):358–383, April 2015.

[109] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, October 2009.

[110] J.S. Poulin, J.M. Caruso, and D.R. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[111] R. Prieto-Diaz. Status report: software reusability. *Software, IEEE*, 10(3):61–66, May 1993.

[112] Al Mamunur Rashid, George Karypis, and John Riedl. Learning preferences of new users in recommender systems: An information theoretic approach. *SIGKDD Explor. Newsl.*, 10(2):90–100, December 2008.

[113] T. Ravichandran and Marcus A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, August 2003.

[114] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, Oct 2001.

[115] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proc. IEEE Int. Conf. on Software Maintenance, 1999.(ICSM'99)*, pages 179–188. IEEE, 1999.

[116] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. Collaborative filtering recommender systems. In *The adaptive web*, pages 291–324. Springer, 2007.

[117] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.

[118] P. Singal, A. K. Mishra, and L. Singh. Test case selection for regression testing of applications using web services based on wsdl specification changes. In *Computing, Communication Automation (ICCCA), 2015 International Conference on*, pages 908–913, May 2015.

[119] Carol Smidts, Chetan Mutha, Manuel Rodríguez, and Matthew J. Gerber. Software testing with an operational profile: Op definition. *ACM Comput. Surv.*, 46(3):39:1–39:39, February 2014.

[120] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. On the danger of coverage directed test case generation. In Juan Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*, pages 409–424. Springer Berlin Heidelberg, 2012.

[121] Huifeng Sun, Yong Peng, Junliang Chen, Chuanchang Liu, and Yuzhuo Sun. A new similarity measure based on adjusted euclidean distance for memory-based collaborative filtering. *Journal of Software (1796217X)*, 6(6), 2011.

[122] Sahar Tahvili, Wasif Afzal, Mehrdad Saadatmand, Markus Bohlin, Daniel Sundmark, and Stig Larsson. *Information Technology: New Generations: 13th International Conference on Information Technology*, chapter Towards Earlier Fault Detection by Value-Driven Prioritization of Test Cases Using Fuzzy TOPSIS, pages 745–759. Springer International Publishing, Cham, 2016.

[123] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product family testing: A survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, March 2004.

[124] R. Torkar and S. Mankefors. A survey on testing and reuse. In *Software: Science, Technology and Engineering, 2003. SwSTE '03. Proceedings. IEEE International Conference on*, pages 164–173, Nov 2003.

[125] W.-T. Tsai, Y. Chen, R. Paul, N. Liao, and H. Huang. Cooperative and group testing in verification of dynamic composite web services. In *Computer Software and Applications Conference COMPSAC 2004. Proc. of the 28th Annual International*, volume 2, pages 170–173 vol.2, Sept 2004.

[126] Wei-Tek Tsai, Yinong Chen, Zhibin Cao, Xiaoying Bai, Hai Huang, and Ray Paul. Testing web services using progressive group testing. In Chi-Hung Chi and Kwok-Yan Lam, editors, *Content Computing*, volume 3309 of *LNCS*, pages 314–322. Springer Berlin Heidelberg, 2004.

[127] J. Voas, L. Morell, and K. Miller. Predicting where faults can hide from testing. *Software, IEEE*, 8(2):41–48, March 1991.

[128] S. Wang, S. Ali, and A. Gotlieb. Cost-effective test suite minimization in product lines using search techniques. *Journal of Systems and Software*, 103:370–391, 2015. cited By 1.

[129] Yi Wei, Bertrand Meyer, and Manuel Oriol. Is branch coverage a good measure of testing effectiveness? In *Emp. Sw. Eng. and Verification*, pages 194–212. Springer, 2012.

[130] S. N. Weiss. Comparing test data adequacy criteria. *SIGSOFT Softw. Eng. Notes*, 14(6):42–49, October 1989.

[131] E.J. Weyuker. Testing component-based software: a cautionary tale. *Software, IEEE*, 15(5):54–59, Sep 1998.

[132] E.J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *Software Engineering, IEEE Transactions on*, 17(7):703–711, 1991.

[133] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *Software Engineering, IEEE Transactions on*, SE-6(3):236–246, 1980.

[134] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set size and block coverage on the fault detection effectiveness. In *Proc. 5th Int. Symposium on Software Reliability Engineering*, pages 230–238, 1994.

[135] M.R. Woodward, D. Hedley, and M.A. Hennell. Experience with path analysis and testing of programs. *Software Engineering, IEEE Transactions on*, SE-6(3):278–286, 1980.

[136] Tao Xie and D. Notkin. Checking inside the black box: regression testing by comparing value spectra. *IEEE Trans. Softw. Eng.*, 31(10):869–883, Oct 2005.

[137] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

[138] Il-Chul Yoon, Alan Sussman, Atif M. Memon, and Adam A. Porter. Testing component compatibility in evolving configurations. *Information & Software Technology*, 55(2):445–458, 2013.

[139] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 214–224, 2015.

[140] Zibin Zheng and Michael R. Lyu. Personalized reliability prediction of web services. *ACM Trans. Softw. Eng. Methodol.*, 22(2):12:1–12:25, March 2013.

[141] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.